

Computer & Informatics
Engineering Department

Technological Educational Institute
of Western Greece

**MSc Technologies and Infrastructures for
Broadband Applications and Services**

MSc Thesis Description

Algebraic Transformations in Computer Graphics

EVANTHIOS PAPADOPOULOS (M13)

Supervisors: I. Kougias, L. Seremeti

ANTIRRIO 2017

Contents

Abstract - Περίληψη	4
Introduction	5
1. The history of computer graphics	6
2. Mathematical background	10
2.1 Vectors.....	10
2.1.1 Vector Notation	10
2.1.2 Graphical Representation of Vectors.....	10
2.1.3 Magnitude of a Vector	12
2.1.4 3D Vectors.....	12
2.1.5 Cartesian Vectors.....	14
2.1.6 Scalar Product	14
2.2 Matrices.....	16
2.2.1 The Determinant of a Matrix	19
2.3 The Laplacian Matrix	20
3. Transformations.....	21
3.1 2D Transformations.....	23
3.1.1 2D Translation.....	23
3.1.2 2D Scaling.....	23
3.1.3 2D Reflections	24
3.1.4 2D Shearing.....	26
3.1.5 2D Rotation	26
3.1.6 2D Scaling using others transformations	28
3.1.7 2D Reflections using others transformations	29
3.2 3D Transformations.....	30
3.2.1 3D Translation.....	30
3.2.2 3D Scaling.....	30
3.2.3 3D Rotations	31
3.2.4 3D Reflections	36
4. Latest Developments.....	37
4.1 Entertainment and Advertising.....	37
4.2 Scientific Visualization	38
4.3 Industrial Design.....	38

5. Matlab	40
6. Design algebraic transformations' algorithms	41
6.1 2-D Transformations Examples	41
6.1.1 Shear Example	42
6.1.2 Scale Example	43
6.1.3 Rotation Example.....	45
6.1.4 Translate Example.....	46
6.2 3-D Transformations Example	48
7. 3D ANIMATION	53
7.1 Create a World Object	53
7.2 Open and View the World	53
7.3 Examine the Virtual World Properties	54
7.4 Finding Nodes of the World	54
7.5 Accessing VRML Nodes	55
7.6 Viewing Fields of Nodes	55
7.7 Moving the Car Node	56
Conclusion	61
References.....	62

Abstract

In this dissertation, we explore the field of computer graphics, its mathematical background, its origins, historical and latest developments. Finally, we present, using Matlab, several examples and interesting applications.

Περίληψη

Σε αυτή τη διατριβή, εξερευνούμε τον τομέα των γραφικών υπολογιστών, το μαθηματικό υπόβαθρο του, τις ρίζες του, τις ιστορικές και τελευταίες εξελίξεις του. Τέλος, παρουσιάζουμε, χρησιμοποιώντας Matlab, διάφορα παραδείγματα και ενδιαφέρουσες εφαρμογές.

Introduction

Computer graphics is a vast and ever expanding field of today's applications, found on television, magazines and newspapers, in all sorts of medical examinations and surgical processes. Industries such as architecture, automotive cartoon and animation that were previously done by hand are now created through computers and, moreover, the rapidly growing industry of video games is perhaps the first one to rely mostly on 3D computer graphics.

The mathematical tools, used in computer graphics programming, are mainly those of linear algebraic transformations, such as *matrices* and *vectors* and, most recently, *Laplacian matrices* are used to develop algorithms for computer graphics and computational photography.

The main purpose of this dissertation is, firstly, to explore the fundamentals of how computers use linear algebraic transformations to create images, computational photos, and videos. Furthermore, in depth research will be conducted on the latest developments of the subject and the origins of computer graphics.

The present work is organized as follows: In the first chapter, the history and the origins of computer graphics is presented, the mathematical background and algebraic transformation are analyzed in the second and third chapters. Next, in chapter 4, the latest developments of the subject in *entertainment* and *advertising*, *scientific visualization*, and *industrial design* areas are explored and, finally, in the last the chapters, with the use of *Matlab* (*chapter 5*), interesting examples of *algebraic transformations in computer graphics* in 2D (*chapter 6*) and 3D animation (*chapter 7*) are given.

1. The history of computer graphics

To understand the many issues in today's modern computer graphics, you need to know how developed computer graphics from its beginnings to this day.

1950 - *Ben Laposky* created the first graphic images, an *Oscilloscope*, generated by an electronic (analog) machine. The image was produced by manipulating electronic beams and recording them onto high-speed film.



1951- The Whirlwind computer at the Massachusetts Institute of Technology was the first computer with a video display of real time data.



1955 - Military applications (SAGE air-defense system used command and control CRT).



1962 - The first graphics station (sketchpad) consisting of a monitor, light pen and software for interactive operation constructed by *Ivan Sutherland*.



" Ivan Sutherland widely regarded as the "*father of computer graphics* "

1964 - Research team working on the algorithms in computer graphics employed at the University of Utah (including Ivan Sutherland, James Blinn, Edwin Catmull).

1965 - The first commercial graphics station: IBM 2250 Display Unit and the IBM PC 1130.



"*The first widely available interactive computer graphics terminal*"

1969 - Beginning of a group SIGGRAPH (Special Interest Group on Graphics) in the organization of ACM (Association for Computing Machinery) gathering of IT professionals.

1974 - Creation of graphics laboratory at the New York Institute of Technology.

1980 - *Turner Whitted* published article about creating realistic images, beginning of method of *ray tracing*.

1982 - *TRON*, the first film that uses computer graphics. The first completely computer-generated scene in the movie *Star Trek II: The Wrath of Khan*.

1983 - Development of fractal techniques and their use in computer graphics. Fractals are used for example in the movie *Star Trek II: The Wrath of Khan*.

1984 - Work of C. Goral, K. Torrance, D. Greenberg and B. Battaile and proposing a new approach for visualization – the method of *radiosity*.

1988 – The first film sequence with morphing in *Willow*.

1989 - The first character created using 3D graphics in the studio Industrial Light & Magic (ILM).

1993 - Dinosaurs in *Jurassic Park* – the first complete and detailed living organisms generated digital technology.

1995 - *Toy Story* implemented complete using computer graphics, the first photo-realistic hair and fur computer generated.



1999 - The first character of the complete human anatomy in a computer-generated studio ILM.

2001 - Photon mapping as the development of ray tracing method.

2006 - Google acquires *Sketchup* (3D modeling software) .

2009 - Film *Avatar* – 3D cinema revolution.



2009 - Decision to create specialty Modern Computer Graphics for Applied Computer Science at the University of Science and Technology in Cracow.

Summing up, the beginnings of computer graphics were related to the military industry, due to the very high cost of their equipment. The development of new graphic techniques and their applications forced the film industry to require, apart from other uses, those of realistic special effects. Currently, computer graphics is also used in many areas of the human life such as photography, industry, sciences, architecture, navigation, cartography, medical diagnostics, special effects in movies, computer games, education, digital art, computational biology and physics, web design, virtual reality etc. [1]

2. Mathematical background

The mathematical tools, used in computer graphics programming, are mainly those of linear algebraic transformations, such as **vectors** and **matrices** and, most recently, **Laplacian matrices** are used to develop algorithms for computer graphics and computational photography. [2]

2.1 Vectors

In computer graphics we employ 2D and 3D vectors. In this chapter we first consider vector notation in a 2D context and then extrapolate the ideas into 3D.

2.1.1 Vector Notation

A scalar such as x is just a name for a single numeric quantity. However, because a vector contains two or more numbers, its symbolic name is printed using a **bold** font to distinguish it from a scalar variable. When a scalar variable is assigned a value we employ the standard algebraic notation

$$x=3$$

However, when a vector is assigned its numeric values, the following notation is used:

$$\mathbf{n} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

which is called a *column* vector. The numbers 3 and 4 are called the *components* of \mathbf{n} , and their position within the brackets is significant. A *row* vector transposes the components horizontally, $\mathbf{n} = [3 \ 4]^T$ where the superscript T reminds us of the transposition.

2.1.2 Graphical Representation of Vectors

Because vectors have to encode direction as well as magnitude, an arrow could be used to indicate direction and a number to specify magnitude. Such a scheme is often used in weather maps. Although this is a useful graphical interpretation for such data, it is not practical for algebraic manipulation. Cartesian coordinates provide an excellent mechanism for visualizing vectors and allowing them to be incorporated within the classical framework of mathematics. Figure 2.1 shows a vector represented by a short line segment.

The length of the line represents the vector's magnitude, and the orientation defines its direction. But as you can see from the figure, the line does not have a direction. Even if we attach an arrowhead to the line, which is standard practice for annotating vectors in books and scientific papers, the arrowhead has no mathematical reality.

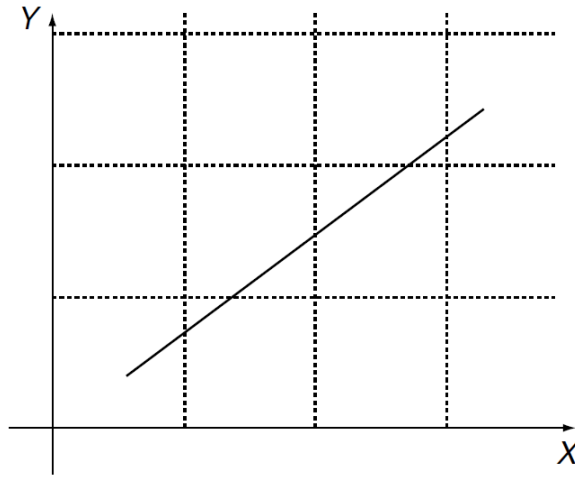


Figure 2.1. A vector represented by a short line segment. However, although the vector has magnitude, it does not have direction.

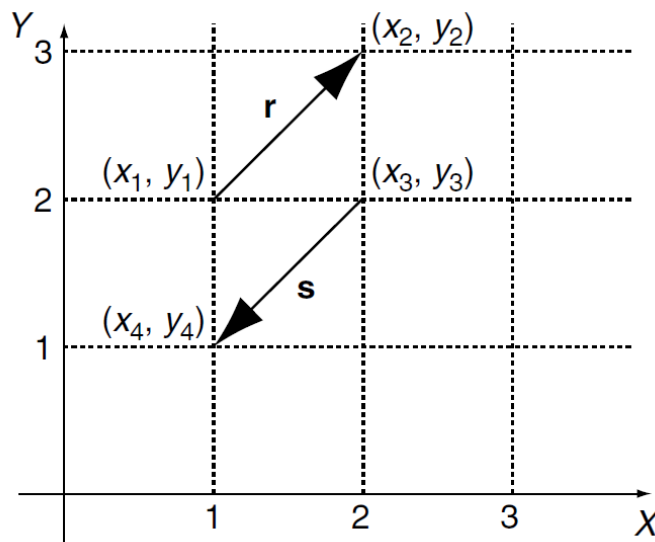


Figure 2.2. Two vectors **r** and **s** have the same magnitude and opposite directions.

The line's direction can be determined by first identifying the vector's tail and then measuring its components along the x - and y -axes. For example, in Figure 3.2 the vector **r** has its tail defined by $(x_1, y_1) = (1, 2)$ and its head by $(x_2, y_2) = (2, 3)$. Vector **s**, on the other hand, has its tail defined by $(x_3, y_3) = (2, 2)$ and its head by $(x_4, y_4) = (1, 1)$. The x - and y -components for **r** are computed as follows:

$$\begin{aligned} x_r &= (x_2 - x_1) & y_r &= (y_2 - y_1) \\ x_r &= 2 - 1 = 1 & y_r &= 3 - 2 = 1 \end{aligned}$$

whereas the components for **s** are computed as follows:

$$\begin{aligned} x_s &= (x_4 - x_3) & y_s &= (y_4 - y_3) \\ x_s &= 1 - 2 = -1 & y_s &= 1 - 2 = -1 \\ x_s &= -1 & y_s &= -1 \end{aligned}$$

It is the negative values of x_s and y_s that encode the vector's direction. In general, given that the coordinates of a vector's head and tail are (x_h, y_h) and (x_t, y_t) respectively, its components Δx and Δy are given by

$$\Delta x = (x_h - x_t) \quad \Delta y = (y_h - y_t)$$

One can readily see from this notation that a vector does not have a unique position in space. It does not matter where we place a vector: so long as we preserve its length and orientation, its components will not alter.

2.1.3 Magnitude of a Vector

The *magnitude* of a vector \mathbf{r} is expressed by $\|\mathbf{r}\|$ and is computed by applying the theorem of Pythagoras to its components:

$$\|\mathbf{r}\| = \sqrt{\Delta x^2 + \Delta y^2}$$

To illustrate these ideas, consider a vector defined by $(x_h, y_h) = (3, 4)$ and $(x_t, y_t) = (1, 1)$. The x - and y -components are 2 and 3 respectively. Therefore its magnitude is equal to

$$\sqrt{2^2 + 3^2} = 3.606$$

Figure 2.3 shows various vectors, and their properties are listed in Table 1.

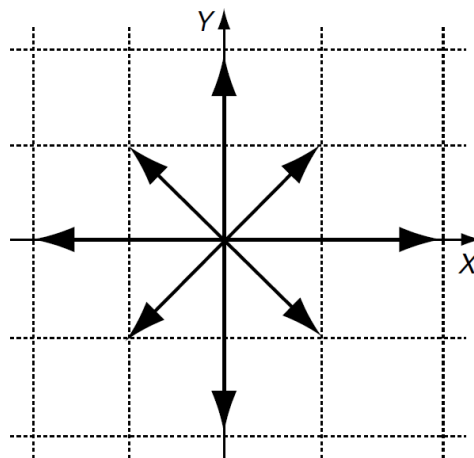


Figure 2.3. Eight vectors, whose coordinates are shown in Table 1

2.1.4 3D Vectors

The above vector examples are in 2D, but it is extremely simple to extend this notation to embrace an extra dimension. Figure 2.4 shows a 3D vector \mathbf{r} with its head, tail, components and magnitude annotated. The components and magnitude are given by

$$\Delta x = (x_h - x_t)$$

x_h	y_h	x_t	y_t	Δx	Δy	$\ \text{Vector}\ $
2	0	0	0	2	0	2
0	2	0	0	0	2	2
-2	0	0	0	-2	0	2
0	-2	0	0	0	-2	2
1	1	0	0	1	1	$\sqrt{2}$
-1	1	0	0	-1	1	$\sqrt{2}$
-1	-1	0	0	-1	-1	$\sqrt{2}$
1	-1	0	0	1	-1	$\sqrt{2}$

Table 1. Values associated with the vectors shown in Figure 3.3

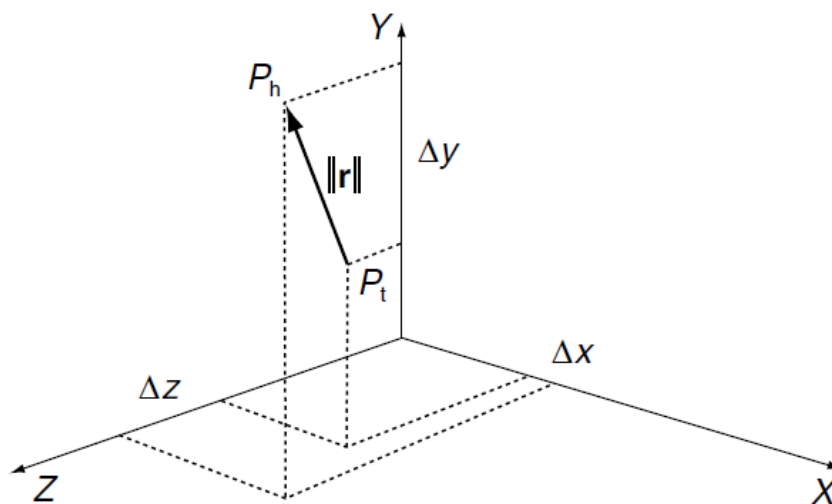


Figure 2.4. The 3D vector has components Δx , Δy , Δz , which are the differences between the head and tail coordinates.

$$\begin{aligned}\Delta y &= (y_h - y_t) \\ \Delta z &= (z_h - z_t) \\ \|\mathbf{r}\| &= \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}\end{aligned}$$

As 3D vectors play a very important part in computer animation.

2.1.5 Cartesian Vectors

Now we can combine the scalar multiplication of vectors, vector addition and unit vectors to permit the algebraic manipulation of vectors. To begin with, we will define three Cartesian unit vectors \mathbf{i} , \mathbf{j} , \mathbf{k} that are aligned with the x -, y - and z -axes respectively:

$$\mathbf{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{j} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Therefore any vector aligned with the x -, y - or z -axes can be defined by a scalar multiple of the unit vectors \mathbf{i} , \mathbf{j} and \mathbf{k} respectively. For example, a vector 10 units long aligned with the x -axis is simply $10\mathbf{i}$, and a vector 20 units long aligned with the z -axis is $20\mathbf{k}$. By employing the rules of vector addition and subtraction, we can compose a vector \mathbf{r} by adding three *Cartesian* vectors as follows:

$$\mathbf{r} = a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$$

This is equivalent to writing \mathbf{r} as

$$\mathbf{r} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

which means that the magnitude of \mathbf{r} is readily computed as

$$\|\mathbf{r}\| = \sqrt{a^2 + b^2 + c^2}$$

Any pair of Cartesian vectors such as \mathbf{r} and \mathbf{s} can be combined as follows:

$$\mathbf{r} = a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$$

$$\mathbf{s} = d\mathbf{i} + e\mathbf{j} + f\mathbf{k}$$

$$\mathbf{r} \pm \mathbf{s} = (a \pm d)\mathbf{i} + (b \pm e)\mathbf{j} + (c \pm f)\mathbf{k}$$

For example, given $\mathbf{r} = 2\mathbf{i} + 3\mathbf{j} + 4\mathbf{k}$ and $\mathbf{s} = 5\mathbf{i} + 6\mathbf{j} + 7\mathbf{k}$

then

$$\mathbf{r} + \mathbf{s} = 7\mathbf{i} + 9\mathbf{j} + 11\mathbf{k}$$

and

$$\|\mathbf{r} + \mathbf{s}\| = \sqrt{7^2 + 9^2 + 11^2} = \sqrt{251} = 15.84$$

2.1.6 Scalar Product

We could multiply two vectors \mathbf{r} and \mathbf{s} by using the product of their magnitudes:

$$\|\mathbf{r}\| \cdot \|\mathbf{s}\|$$

Although this is a valid operation, it does not get us anywhere because it ignores the orientation of the vectors, which is one of their important features. The concept, however, is readily developed into a useful operation by including the angle between the vectors. *Figure 2.5* shows two vectors \mathbf{r} and \mathbf{s} that have been drawn, for convenience, such that their tails touch. Taking \mathbf{s} as the reference vector, which is an arbitrary choice, we compute the projection of \mathbf{r} on \mathbf{s} , which takes into account their relative orientation. The length of \mathbf{r} on \mathbf{s} is $\|\mathbf{r}\| \cos(\beta)$. We can now multiply the magnitude of \mathbf{s} by the projected length of \mathbf{r} :

$$\|\mathbf{s}\| \cdot \|\mathbf{r}\| \cos(\beta)$$

This scalar product is written

$$\mathbf{s} \cdot \mathbf{r} = ||\mathbf{s}|| \cdot ||\mathbf{r}|| \cos(\beta)$$

The dot symbol ‘ \cdot ’ is used to represent scalar multiplication, to distinguish it from the vector product, which employs an ‘ \times ’ symbol.

Because of this symbol, the scalar product is often referred to as the dot product.

So far we have only defined what we mean by the dot product. We now need to find out how to compute it. Fortunately, everything is in place to perform this task. To begin with, we define two Cartesian vectors \mathbf{r} and \mathbf{s} , and proceed to multiply them together using the dot product definition:

$$\mathbf{r} = a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$$

$$\mathbf{s} = d\mathbf{i} + e\mathbf{j} + f\mathbf{k}$$

therefore

$$\begin{aligned} \mathbf{r} \cdot \mathbf{s} &= (a\mathbf{i} + b\mathbf{j} + c\mathbf{k}) \cdot (d\mathbf{i} + e\mathbf{j} + f\mathbf{k}) \\ &= a\mathbf{i} \cdot (d\mathbf{i} + e\mathbf{j} + f\mathbf{k}) + b\mathbf{j} \cdot (d\mathbf{i} + e\mathbf{j} + f\mathbf{k}) + c\mathbf{k} \cdot (d\mathbf{i} + e\mathbf{j} + f\mathbf{k}) \end{aligned}$$

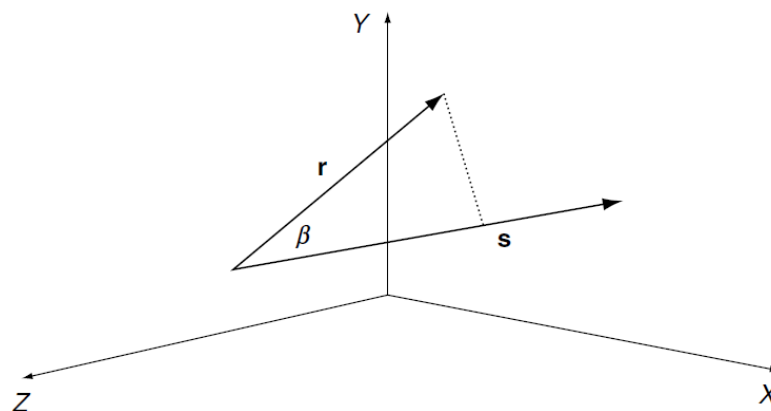


Figure 2.5. The projection of \mathbf{r} on \mathbf{s} creates the basis for the scalar product

$$\begin{aligned} \mathbf{r} \cdot \mathbf{s} &= ad(\mathbf{i} \cdot \mathbf{i}) + ae(\mathbf{i} \cdot \mathbf{j}) + af(\mathbf{i} \cdot \mathbf{k}) + \\ &\quad bd(\mathbf{j} \cdot \mathbf{i}) + be(\mathbf{j} \cdot \mathbf{j}) + bf(\mathbf{j} \cdot \mathbf{k}) + \\ &\quad cd(\mathbf{k} \cdot \mathbf{i}) + ce(\mathbf{k} \cdot \mathbf{j}) + cf(\mathbf{k} \cdot \mathbf{k}) \end{aligned}$$

Before we proceed any further, we can see that we have created various dot product terms such as $(\mathbf{i} \cdot \mathbf{i})$, $(\mathbf{j} \cdot \mathbf{j})$, $(\mathbf{k} \cdot \mathbf{k})$, etc. These terms can be divided into two groups: those that involve the same unit vector, and those that reference different unit vectors.

Using the definition of the dot product, terms such as $(\mathbf{i} \cdot \mathbf{i})$, $(\mathbf{j} \cdot \mathbf{j})$ and $(\mathbf{k} \cdot \mathbf{k}) = 1$, because the angle between \mathbf{i} and \mathbf{i} , \mathbf{j} and \mathbf{j} , or \mathbf{k} and \mathbf{k} is 0° ; and $\cos(0^\circ) = 1$. But because the other vector combinations are separated by 90° , and $\cos(90^\circ) = 0$, all remaining terms collapse to zero. Bearing in mind that the magnitude of a unit vector is 1, we can write

$$||\mathbf{s}|| \cdot ||\mathbf{r}|| \cos(\beta) = ad + be + cf$$

This result confirms that the dot product is indeed a scalar quantity.

2.2 Matrices

Matrix notation was investigated by the British mathematician Arthur Cayley around 1858. Cayley formalized matrix algebra, along with the American mathematicians Benjamin and Charles Pierce. Also, by the start of the 19th century Carl Gauss (1777–1855) had proved that transformations were not commutative, i.e. $T_1 \times T_2 \neq T_2 \times T_1$, and Cayley's matrix notation would clarify such observations. For example, consider the transformation T_1 :

$$\mathbf{T}_1 \begin{array}{l} x' = ax + by \\ y' = cx + dy \end{array}$$

and another transformation T_2 that transforms T_1 :

$$\mathbf{T}_2 \times \mathbf{T}_1 \begin{array}{l} x'' = Ax' + By' \\ y'' = Cx' + Dy' \end{array}$$

If we substitute the full definition of T_1 we get

$$\mathbf{T}_2 \times \mathbf{T}_1 \begin{array}{l} x'' = A(ax + by) + B(cx + dy) \\ y'' = C(ax + by) + D(cx + dy) \end{array}$$

which simplifies to

$$\mathbf{T}_2 \times \mathbf{T}_1 \begin{array}{l} x'' = (Aa + Bc)x + (Ab + Bd)y \\ y'' = (Ca + Dc)x + (Cb + Dd)y \end{array}$$

Caley proposed separating the constants from the variables, as follows:

$$\mathbf{T}_1 \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

where the square matrix of constants in the middle determines the transformation.

The algebraic form is recreated by taking the top variable x' , introducing the = sign, and multiplying the top row of constants $[a \ b]$ individually by the last column vector containing x and y . We then examine the second variable y' , introduce the = sign, and multiply the bottom row of constants $[c \ d]$ individually by the last *column* vector containing x and y , to create

$$\begin{array}{l} x' = ax + by \\ y' = cx + dy \end{array}$$

Using Caley's notation, the product $T_2 \times T_1$ is

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} x' \\ y' \end{bmatrix}$$

But the notation also intimated that

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

and when we multiply the two *inner matrices* together they must produce

$$\begin{aligned} x'' &= (Aa + Bc)x + (Ab + Bd)y \\ y'' &= (Ca + Dc)x + (Cb + Dd)y \end{aligned}$$

or in matrix form

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} Aa + Bc & Ab + Bd \\ Ca + Dc & Cb + Dd \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

otherwise the two systems of notation will be inconsistent. This implies that

$$\begin{bmatrix} Aa + Bc & Ab + Bd \\ Ca + Dc & Cb + Dd \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

which demonstrates how matrices must be multiplied.

Here are the rules for matrix multiplication:

$$\begin{array}{|c|c|} \hline \boxed{Aa+Bc} & \\ \hline & \\ \hline \end{array} = \begin{array}{|c|c|} \hline \boxed{A} & \boxed{B} \\ \hline & \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline \boxed{a} & \\ \hline \boxed{c} & \\ \hline \end{array}$$

1 The top left-hand corner element $Aa+Bc$ is the product of the top row of the first matrix by the left column of the second matrix.

$$\begin{array}{|c|c|} \hline & \boxed{Ab+Bd} \\ \hline & \\ \hline \end{array} = \begin{array}{|c|c|} \hline \boxed{A} & \boxed{B} \\ \hline & \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline & \boxed{b} \\ \hline & \boxed{d} \\ \hline \end{array}$$

2 The top right-hand element $Ab + Bd$ is the product of the top row of the first matrix by the right column of the second matrix.

$$\begin{array}{|c|c|} \hline & \\ \hline Ca+Dc & \\ \hline \end{array} = \begin{array}{|c|c|} \hline & \\ \hline C & D \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline a & \\ \hline c & \\ \hline \end{array}$$

3 The bottom left-hand element $Ca + Dc$ is the product of the bottom row of the first matrix by the left column of the second matrix.

$$\begin{array}{|c|c|} \hline & \\ \hline & Cb+Dd \\ \hline \end{array} = \begin{array}{|c|c|} \hline & \\ \hline C & D \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline & b \\ \hline & d \\ \hline \end{array}$$

4 The bottom right-hand element $Cb+Dd$ is the product of the bottom row of the first matrix by the right column of the second matrix.

It is now a trivial exercise to confirm Gauss's observation that $T1 \times T2 \neq T2 \times T1$, because if we reverse the transforms $T2 \times T1$ to $T1 \times T2$ we get

$$\begin{bmatrix} Aa + Bc & Ab + Bd \\ Ca + Dc & Cb + Dd \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

which shows conclusively that the product of two transforms is not commutative.

One immediate problem with this notation is that there is no apparent mechanism to add or subtract a constant such as c or f :

$$\begin{aligned} x' &= ax + by + c \\ y' &= dx + ey + f \end{aligned}$$

Mathematicians resolved this in the 19th century, by the use of *homogeneous coordinates*.

Basically, *homogeneous coordinates* define a point in a plane using three coordinates instead of two. The reason why this coordinate system is called 'homogeneous' is because it is possible to transform functions such as $f(x, y)$ into the form $f(x/t, y/t)$ without disturbing the degree of the curve. To the non-mathematician this may not seem anything to get excited about, but in the field of projective geometry it is a very powerful concept.

For our purposes, we can imagine that a collection of homogeneous points of the form (x, y, t) exist on an xy -plane where t is the z -coordinate, as illustrated in *Figure 2.6*. The figure shows a triangle on the $t = 1$ plane, and a similar triangle, much larger, on a more distant plane. Thus instead of working in two dimensions, we can work on an arbitrary xy -plane in three dimensions. The t - or z -coordinate of the plane is immaterial because the x - and y -coordinates are eventually scaled by t . However, to keep things simple it seems a good idea to choose $t = 1$. This means that the point (x, y) has homogeneous coordinates $(x, y, 1)$, making scaling unnecessary.

If we substitute 3D homogeneous coordinates for traditional 2D Cartesian coordinates, we must attach a 1 to every (x, y) pair. When a point $(x, y, 1)$ is transformed, it will emerge as $(x, y, 1)$, and we discard the 1. This may seem a futile exercise, but it resolves the problem of creating a translation transformation.

Consider the following transformation on the homogeneous point $(x, y, 1)$:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This expands to

$$\begin{aligned} x' &= ax + by + c \\ y' &= dx + ey + f \\ 1 &= 1 \end{aligned}$$

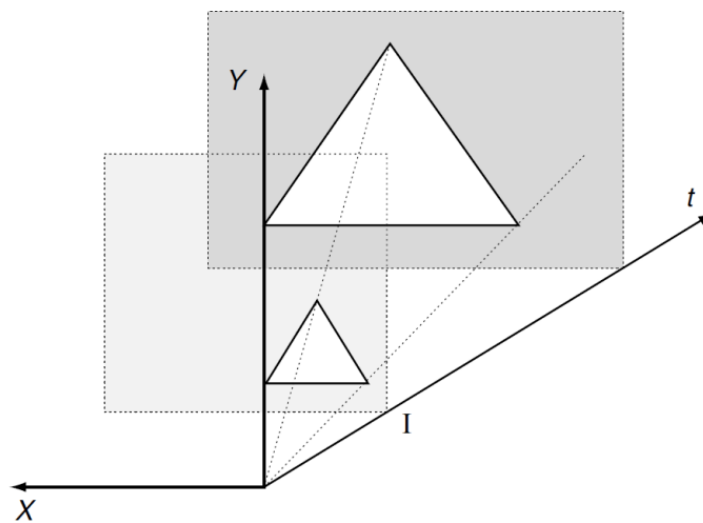


Figure 2.6. 2D homogeneous coordinates can be visualized as a plane in 3D space, generally where $t = 1$, for convenience.

2.2.1 The Determinant of a Matrix

The *determinant* of a 2×2 matrix is a scalar quantity computed. Given a matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

its determinant is $ad - cb$ and is represented by

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

For example, the determinant of

$$\begin{bmatrix} 3 & 2 \\ 1 & 2 \end{bmatrix} \text{ is } 3 \times 2 - 1 \times 2 = 4$$

Later, we will discover that the determinant of a 2×2 matrix determines the change in area that occurs when a polygon is transformed by the matrix.

For example, if the determinant is 1, there is no change in area, but if the determinant is 2, the polygon's area is doubled.

2.3 The Laplacian Matrix

The *Laplacian matrix*, sometimes also called the *admittance matrix* or *Kirchhoff matrix*, of a graph G , where $G=(V, E)$ is an undirected, unweighted graph without graph loops (i, i) or multiple edges from one node to another, V is the vertex set, $n=|V|$, and E is the edge set, is an $n \times n$ symmetric matrix with one row and column for each node defined by $L = D - A$

where $D = \text{diag}(d_1, \dots, d_n)$ is the degree matrix, which is the diagonal matrix formed from the vertex degrees and A is the adjacency matrix. The diagonal elements $L_{i,j}$ of L are therefore equal the degree of vertex v_i and off-diagonal elements $L_{i,j}$ are -1 if vertex v_i is adjacent to v_j and 0 otherwise.

A normalized version of the Laplacian matrix, denoted \mathcal{L} , is similarly defined by

$$\mathcal{L}_{i,j}(G) = \begin{cases} 1 & \text{if } i = j \text{ and } d_j \neq 0 \\ -\frac{1}{\sqrt{d_i d_j}} & \text{if } i \text{ and } j \text{ are adjacent} \\ 0 & \text{otherwise} \end{cases}$$

The Laplacian matrix is a discrete analog of the Laplacian operator in multivariable calculus and serves a similar purpose by measuring to what extent a graph differs at one vertex from its values at nearby vertices. The Laplacian matrix arises in the analysis of random walks and electrical networks on graphs, and in particular in the computation of resistance distances.

[3]

3. Transformations

Transformations are used to **scale**, **translate**, **rotate**, **reflect** and **shear** shapes and objects. And, as we shall discover shortly, it is possible to affect this by changing their coordinate values. Although algebra is the basic notation for transformations, it is also possible to express them as *matrices*, which provide certain advantages for viewing the transformation and for interfacing to various types of computer graphics hardware. [2]

Translation

Cartesian coordinates provide a one-to-one relationship between number and shape, such that when we change a shape's coordinates, we change its geometry. For example, if $P(x, y)$ is a vertex on a shape, when we apply the operation $x = x + 3$ we create a new point $P(x, y)$ three units to the right. Similarly, the operation $y = y + 1$ creates a new point $P(x, y)$ displaced one unit vertically. By applying both of these transforms to every vertex to the original shape, the shape is displaced as shown in *Figure 3.1*.

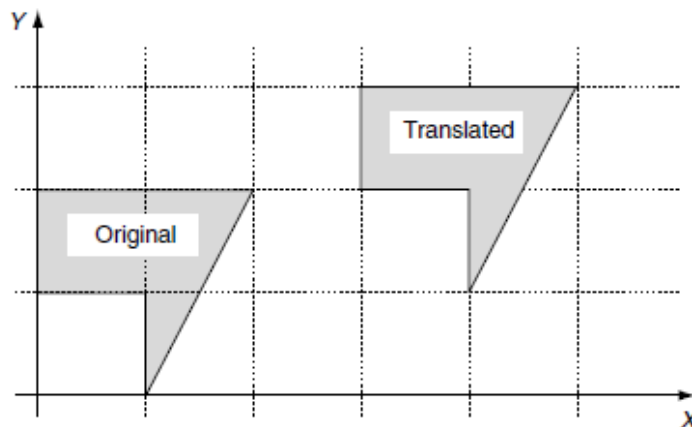


Figure 3.1. The translated shape results by adding 3 to every x-coordinate, and 1 to every y-coordinate of the original shape.

Scaling

Shape scaling is achieved by multiplying coordinates as follows:

$$\begin{aligned}x' &= 2x \\y' &= 1.5y\end{aligned}$$

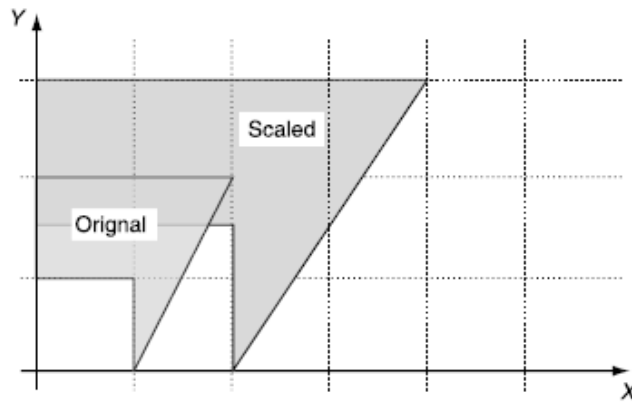


Figure 3.2 The scaled shape results by multiplying every x -coordinate by 2 and every y -coordinate by 1.5.

This transform results in a horizontal scaling of 2 and a vertical scaling of 1.5, as illustrated in *Figure 3.2*. Note that a point located at the origin does not change its place, so scaling is relative to the origin.

Reflection

To make a reflection of a shape relative to the y -axis, we simply reverse the sign of the x -coordinate, leaving the y -coordinate unchanged

$$\begin{aligned}x' &= -x \\ y' &= y\end{aligned}$$

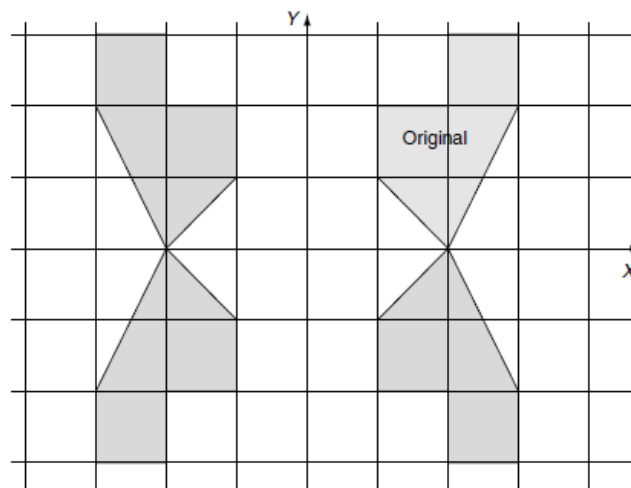


Figure 3.3. The top right-hand shape can give rise to the three reflections simply by reversing the signs of coordinates.

and to reflect a shape relative to the x -axis we reverse the y -coordinates:

$$\begin{aligned}x' &= x \\ y' &= -y\end{aligned}$$

3.1 2D Transformations

3.1.1 2D Translation

The algebraic and matrix notation for 2D translation is

$$\begin{aligned}x' &= x + t_x \\y' &= y + t_y\end{aligned}$$

or, using matrices,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3.1.2 2D Scaling

The algebraic and matrix notation for 2D scaling is

$$\begin{aligned}x' &= s_x x \\y' &= s_y y\end{aligned}$$

or, using matrices,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The scaling action is relative to the origin, i.e. the point (0,0) remains (0,0). All other points move away from the origin. To scale relative to another point (p_x, p_y) we first subtract (p_x, p_y) from (x, y) respectively. This effectively translates the reference point (p_x, p_y) back to the origin. Second, we perform the scaling operation, and third, add (p_x, p_y) back to (x, y) respectively, to compensate for the original subtraction. Algebraically this is

$$\begin{aligned}x' &= s_x(x - p_x) + p_x \\y' &= s_y(y - p_y) + p_y\end{aligned}$$

which simplifies to

$$\begin{aligned}x' &= s_x x + p_x(1 - s_x) \\y' &= s_y y + p_y(1 - s_y)\end{aligned}$$

or in a homogeneous matrix form

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & p_x(1 - s_x) \\ 0 & s_y & p_y(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (1)$$

For example, to scale a shape by 2 relative to the point (1, 1) the matrix is

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3.1.3 2D Reflections

The matrix notation for reflecting about the y -axis is:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

or about the x -axis

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

However, to make a reflection about an arbitrary vertical or horizontal axis we need to introduce some more algebraic deception. For example, to make a reflection about the vertical axis $x = 1$, we first subtract 1 from the x -coordinate. This effectively makes the $x = 1$ axis coincident with the major y -axis. Next we perform the reflection by reversing the sign of the modified x -coordinate. And finally, we add 1 to the reflected coordinate to compensate for the original subtraction. Algebraically, the three steps are

$$\begin{aligned} x_1 &= x - 1 \\ x_2 &= -(x - 1) \\ x' &= -(x - 1) + 1 \end{aligned}$$

which simplifies to

$$\begin{aligned} x' &= -x + 2 \\ y' &= y \end{aligned}$$

or in matrix form,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 3.4 illustrates this process.

In general, to reflect a shape about an arbitrary y -axis, $y=a_x$, the following transform is required:

$$\begin{aligned} x' &= -(x - a_x) + a_x = -x + 2a_x \\ y' &= y \end{aligned}$$

or, in matrix form,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 2a_x \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2)$$

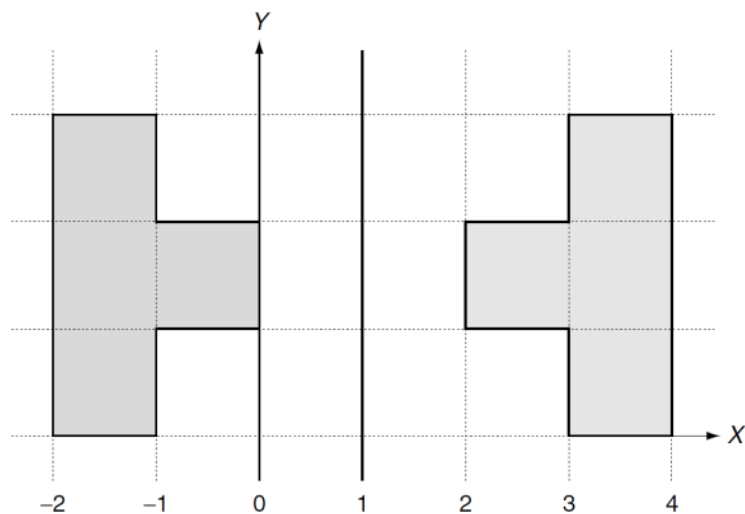


Figure 3.4. The shape on the right is reflected about the $x = 1$ axis.

Similarly, this transform is used for reflections about an arbitrary x -axis, $y = a_y$:

$$\begin{aligned} x' &= x \\ y' &= -(y - a_y) + a_y = -y + 2a_y \end{aligned}$$

or, in matrix form,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2a_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3.1.4 2D Shearing

A shape is sheared by leaning it over at an angle β . *Figure 3.5* illustrates the geometry, and we see that the y -coordinate remains unchanged but the x -coordinate is a function of y and $\tan(\beta)$.

$$x' = x + y \tan(\beta)$$

$$y' = y$$

or, in matrix form,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \tan(\beta) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

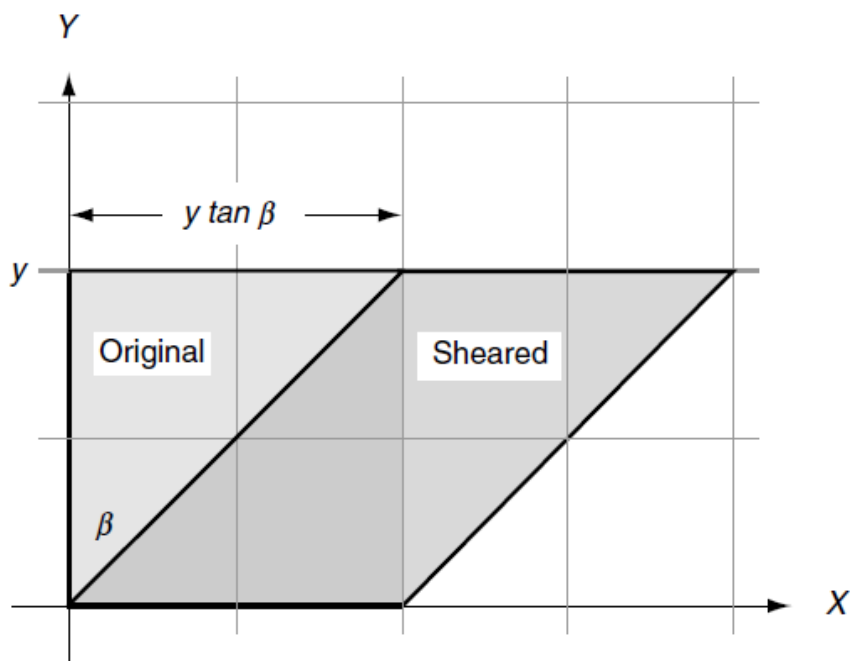


Figure 3.5. The original square shape is sheared to the right by an angle β , and the horizontal shift is proportional to $y \tan(\beta)$.

3.1.5 2D Rotation

Figure 3.6 shows a point $P(x, y)$ which is to be rotated by an angle β about the origin to $P'(x', y')$. It can be seen that

$$x' = R \cos(\theta + \beta)$$

$$y' = R \sin(\theta + \beta)$$

therefore,

$$x' = R(\cos(\theta) \cos(\beta) - \sin(\theta) \sin(\beta))$$

$$y' = R(\sin(\theta) \cos(\beta) + \cos(\theta) \sin(\beta))$$

$$x' = R \left(\frac{x}{R} \cos(\beta) - \frac{y}{R} \sin(\beta) \right)$$

$$y' = R \left(\frac{y}{R} \cos(\beta) + \frac{x}{R} \sin(\beta) \right)$$

$$x' = x \cos(\beta) - y \sin(\beta)$$

$$y' = x \sin(\beta) + y \cos(\beta)$$

or, in matrix form,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\beta) & -\sin(\beta) & 0 \\ \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

For example, to rotate a point by 90° the matrix becomes

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

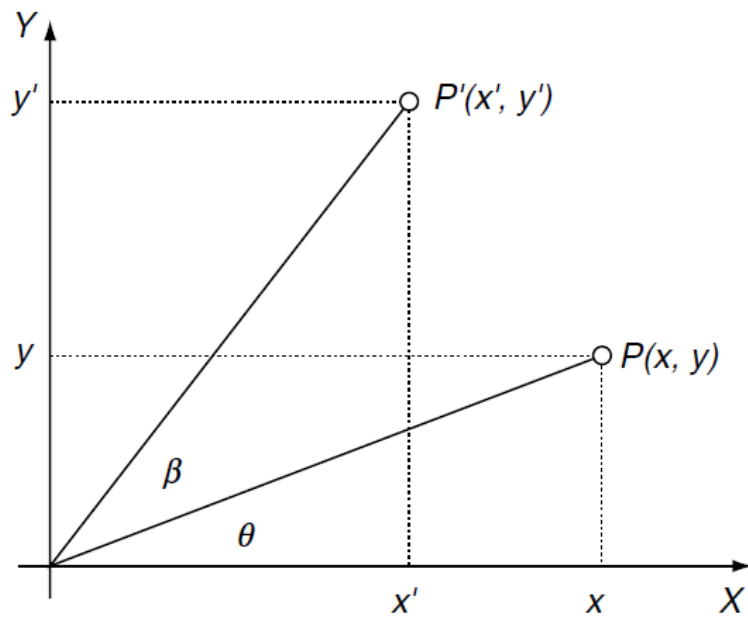


Figure 3.6. The point $P(x, y)$ is rotated through an angle β to $P(x', y')$.

Thus the point (1, 0) becomes (0, 1). If we rotate by 360° the matrix becomes

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Such a matrix has a null effect and is called an *identity matrix*.

3.1.6 2D Scaling using others transformations

The strategy we used to scale a point (x, y) relative to some arbitrary point (p_x, p_y) was to first, translate (-p_x, -p_y); second, perform the scaling; and third, translate (p_x, p_y). These three transforms can be represented in matrix form as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = [\text{translate}(p_x, p_y)] \cdot [\text{scale}(s_x, s_y)] \cdot [\text{translate}(-p_x, -p_y)] \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

which expands to

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -p_x \\ 0 & 1 & -p_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Note the sequence of the transforms, as this often causes confusion. The first transform acting on the point (x, y, 1) is translate (-p_x, -p_y), followed by scale (s_x, s_y), followed by translate (p_x, p_y). If they are placed in any other sequence, you will discover, like Gauss, that transforms are not commutative!

We can now concatenate these matrices into a single matrix by multiplying them together. This can be done in any sequence, so long as we preserve the original order. Let's start with scale (s_x, s_y) and translate (-p_x, -p_y). This produces

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & -s_x p_x \\ 0 & s_y & -s_y p_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

and finally

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & p_x(1 - s_x) \\ 0 & s_y & p_y(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

which is the same as the previous transform (1).

3.1.7 2D Reflections using others transformations

A reflection about the y -axis is given by

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Therefore, using matrices, we can reason that a reflection transform about an arbitrary axis $x = a_x$, parallel with the y -axis, is given by

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = [\text{translate}(a_x, 0)] \cdot [\text{reflection}] \cdot [\text{translate}(-a_x, 0)] \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

which expands to

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a_x \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -a_x \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

We can now concatenate these matrices into a single matrix by multiplying them together. Let's begin by multiplying the reflection and the translate $(-a_x, 0)$ matrices together. This produces

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a_x \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & a_x \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

and finally

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 2a_x \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

which is the same as the previous transform (2).

3.2 3D Transformations

Now we come to transformations in **three dimensions**, where we apply the same reasoning as in two dimensions. Scaling and translation are basically the same, but where in 2D we rotated a shape about *a point*, in 3D we rotate an object about *an axis*.

3.2.1 3D Translation

The algebra is so simple for 3D translation that we can write the homogeneous matrix directly:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3.2.2 3D Scaling

The algebra for 3D scaling is

$$\begin{aligned} x' &= s_x x \\ y' &= s_y y \\ z' &= s_z z \end{aligned}$$

which in matrix form is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The scaling is relative to the origin, but we can arrange for it to be relative to an arbitrary point (p_x, p_y, p_z) with the following algebra:

$$\begin{aligned} x' &= s_x(x - p_x) + p_x \\ y' &= s_y(y - p_y) + p_y \\ z' &= s_z(z - p_z) + p_z \end{aligned}$$

which in matrix form is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & p_x(1 - s_x) \\ 0 & s_y & 0 & p_y(1 - s_y) \\ 0 & 0 & s_z & p_z(1 - s_z) \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3.2.3 3D Rotations

In two dimensions a shape is rotated about a point, whether it is the origin or some arbitrary position. In three dimensions an object is rotated about an axis, whether it is the x -, y - or z -axis, or some arbitrary axis. To begin with, let's look at rotating a vertex about one of the three orthogonal axes; such rotations are called *Euler rotations* after the Swiss mathematician Leonhard Euler (1707–1783).

Recall that a general 2D-rotation transform is given by

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\beta) & -\sin(\beta) & 0 \\ \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

which in 3D can be visualized as rotating a point $P(x, y, z)$ on a plane parallel with the xy -plane as shown in *Figure 3.7*. In algebraic terms this can be written as

$$\begin{aligned} x' &= x \cos(\beta) - y \sin(\beta) \\ y' &= x \sin(\beta) + y \cos(\beta) \\ z' &= z \end{aligned}$$

Therefore, the 3D transform can be written as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\beta) & -\sin(\beta) & 0 & 0 \\ \sin(\beta) & \cos(\beta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

which basically rotates a point about the z -axis.

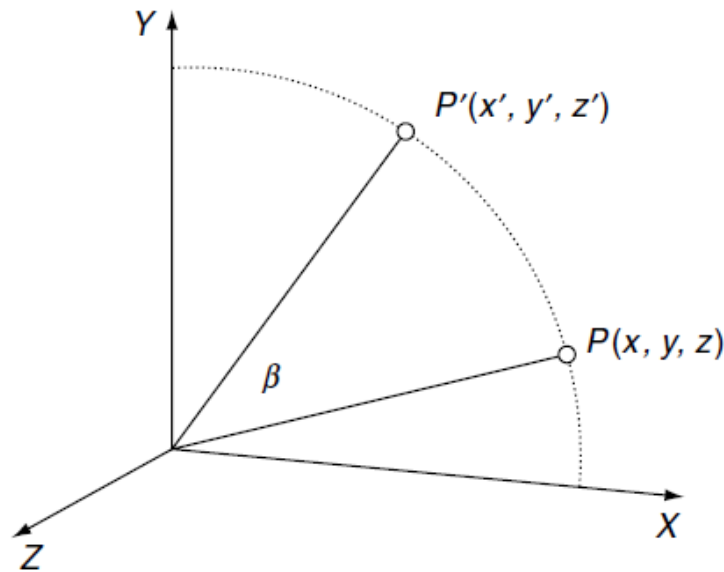


Figure 3.7. Rotating P about the z -axis.

When rotating about the x -axis, the x -coordinate remains constant while the y - and z -coordinates are changed. Algebraically, this is

$$\begin{aligned}x' &= x \\y' &= y \cos(\beta) - z \sin(\beta) \\z' &= y \sin(\beta) + z \cos(\beta)\end{aligned}$$

or, in matrix form

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\beta) & -\sin(\beta) & 0 \\ 0 & \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

When rotating about the y -axis, the y -coordinate remains constant while the x - and z -coordinates are changed. Algebraically, this is

$$\begin{aligned}x' &= z \sin(\beta) + x \cos(\beta) \\y' &= y \\z' &= z \cos(\beta) - x \sin(\beta)\end{aligned}$$

or, in matrix form

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Note that the matrix terms do not appear to share the symmetry seen in the previous two matrices. Nothing has really gone wrong, it is just the way the axes are paired together to rotate the coordinates.

The above rotations are also known as **yaw**, **pitch** and **roll**. Great care should be taken with these terms when referring to other books and technical papers. Sometimes a left-handed system of axes is used rather than a right-handed set, and the vertical axis may be the y -axis or the z -axis.

Consequently, the matrices representing the rotations can vary greatly. In this text all Cartesian coordinate systems are right-handed, and the vertical axis is always the y -axis.

The roll, pitch and yaw angles can be defined as follows:

- **roll** is the angle of rotation about the **z -axis**
- **pitch** is the angle of rotation about the **x -axis**
- **yaw** is the angle of rotation about the **y -axis**

Figure 3.8 illustrates these rotations and the sign convention. The homogeneous matrices representing these rotations are as follows:

- rotate *roll* about the z -axis:

$$\begin{bmatrix} \cos(\text{roll}) & -\sin(\text{roll}) & 0 & 0 \\ \sin(\text{roll}) & \cos(\text{roll}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- rotate *pitch* about the x -axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\text{pitch}) & -\sin(\text{pitch}) & 0 \\ 0 & \sin(\text{pitch}) & \cos(\text{pitch}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- rotate *yaw* about the y -axis:

$$\begin{bmatrix} \cos(\text{yaw}) & 0 & \sin(\text{yaw}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\text{yaw}) & 0 & \cos(\text{yaw}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

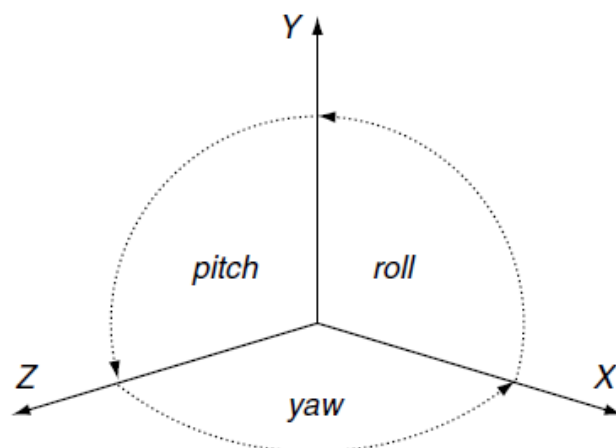


Figure 3.8. The convention for *roll*, *pitch* and *yaw* angles.

A common sequence for applying these rotations is *roll*, *pitch*, *yaw*, as seen in the following transform:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = [yaw] \cdot [pitch] \cdot [roll] \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and if a translation is involved,

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = [translate] \cdot [yaw] \cdot [pitch] \cdot [roll] \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

When these rotation transforms are applied, the vertex is first rotated about the *z*-axis (roll), followed by a rotation about the *x*-axis (pitch), followed by a rotation about the *y*-axis (yaw). Euler rotations are relative to the fixed frame of reference. This is not always easy to visualize, as one's attention is normally with the rotating frame of reference.

Let's consider a simple example where an axial system is subjected to a pitch rotation followed by a yaw rotation relative to fixed frame of reference. We begin with two frames of reference *XYZ* and *X'Y'Z'* mutually aligned. *Figure 3.9* shows the orientation of *X'Y'Z'* after it is subjected to a pitch of 90° about the *x*-axis. *Figure 3.10* shows the final orientation after *X'Y'Z'* is subjected to a yaw of 90° about the *y*-axis.

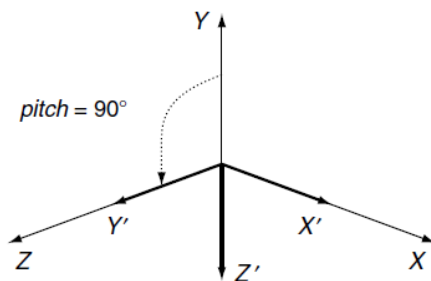


Fig. 3.9. The *X'Y'Z'* axial system after a *pitch* of 90°.

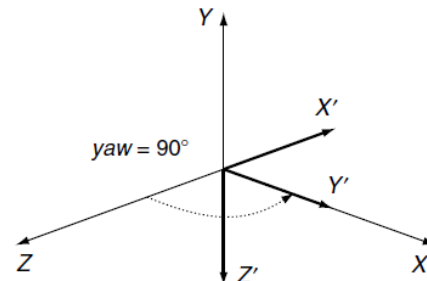


Fig.3.10. The *X'Y'Z'* axial system after a *yaw* of 90°.

Rotating about an Axis

The above rotations were relative to the *x*-, *y*- and *z*-axes. Now let's consider rotations about an axis parallel to one of these axes. To begin with, we will rotate about an axis parallel with the *z*-axis, as shown in *Figure 3.11*.

The scenario is very reminiscent of the 2D case for rotating a point about an arbitrary point, and the general transform is given by

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = [translate(p_x, p_y, 0)] \cdot [rotate\beta] \cdot [translate(-p_x, -p_y, 0)] \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and the matrix is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\beta) & -\sin(\beta) & 0 & p_x(1 - \cos(\beta)) + p_y \sin(\beta) \\ \sin(\beta) & \cos(\beta) & 0 & p_y(1 - \cos(\beta)) - p_x \sin(\beta) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

I hope you can see the similarity between rotating in 3D and 2D: the x - and y -coordinates are updated while the z -coordinate is held constant.

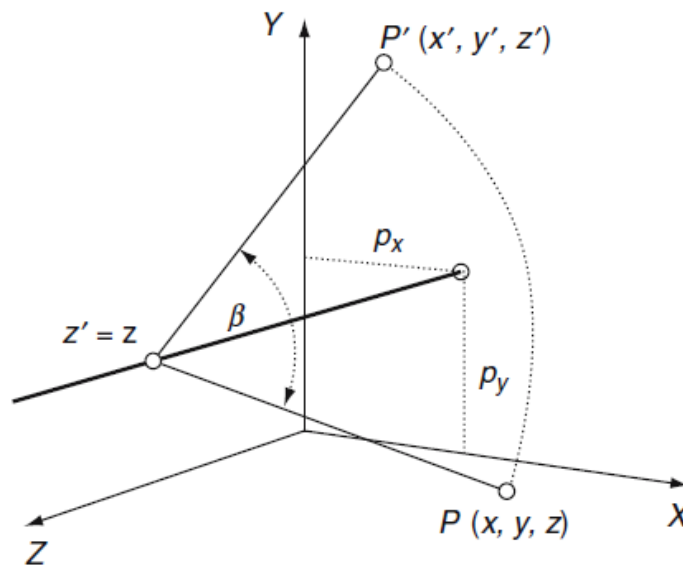


Figure 3.11. Rotating a point about an axis parallel with the z -axis

We can now state the other two matrices for rotating about an axis parallel with the x -axis and parallel with the y -axis:

- rotating about an axis parallel with the x -axis:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\beta) & -\sin(\beta) & p_y(1 - \cos(\beta)) + p_z \sin(\beta) \\ 0 & \sin(\beta) & \cos(\beta) & p_z(1 - \cos(\beta)) - p_y \sin(\beta) \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- rotating about an axis parallel with the y -axis:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & p_x(1 - \cos(\beta)) - p_z \sin(\beta) \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & p_z(1 - \cos(\beta)) + p_x \sin(\beta) \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3.2.4 3D Reflections

Reflections in 3D occur with respect to a plane, rather than an axis. The matrix giving the reflection relative to the yz -plane is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and the reflection relative to a plane parallel to, and a_x units from, the yz - plane is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 2a_x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4. Latest Developments

Computer graphics, a subfield of computer science, is concerned with digitally synthesizing and manipulating visual content. Although the term often refers to three-dimensional (3D) computer graphics, it also encompasses two-dimensional (2D) graphics and image processing. Graphics is often differentiated from the field of visualization, although the two have many similarities. *Entertainment* (in the form of animated movies and video games) is perhaps the most well-known application of computer graphics.

Today, computer graphics can be seen in almost every illustration made. Computer graphics are often used by photographers to improve *photos*. It also has many other applications, ranging from the motion picture industry to architectural rendering. As a tool, computer graphics, which were once very expensive and complicated, can now be used by anyone in the form of freeware. In the future, computer graphics could possibly replace traditional drawing or painting for illustrations. Already, it is being used as a form of enhancement for different illustrations.

The use of and relevance of computer graphics has blossomed in many areas in the past 20 years, ranging from the studio arts to new mathematical disciplines such as computational geometry. The areas, in which graphics have arguably had the most impact,—and certainly the most visibility—can loosely be categorized as *entertainment* and *advertising*, *scientific visualization*, and *industrial design* [4].

4.1 Entertainment and Advertising

No doubt the most stylish deployment of computer graphics today is in Hollywood and on Madison Avenue. *Special effects*, *photographic manipulations*, *computer animation*, and other digital trickery routinely spice up (often otherwise dull) movies and *ad spots*. Students are aware that many of these effects—based as they are on generating shapes and transforming shapes over time—are inherently geometric in nature. From the perspective of classroom geometry, these graphics applications can be great motivators.



Figure 4.1 3D animation movie (2017)

4.2 Scientific Visualization

Though slightly less glamorous than Hollywood, scientific visualization forms a second important focus of computational modeling and graphics efforts. Here, computer-generated illustrations and simulations are used to depict the structure of objects that cannot otherwise be inspected because they are too *small* (e.g., chemical compounds and crystal structures), too *large* (global weather patterns), too *remote* (topography of distant planets), too abstract (such as multi-dimensional mathematical manifolds), or too *dangerous* (such as atmospheric conditions in the eye of a hurricane and in deep ocean trenches). In fact, most computer graphics technologies are originally developed to provide some new tools to the scientific visualization community, and then later reappear in less expensive applications within other domains.

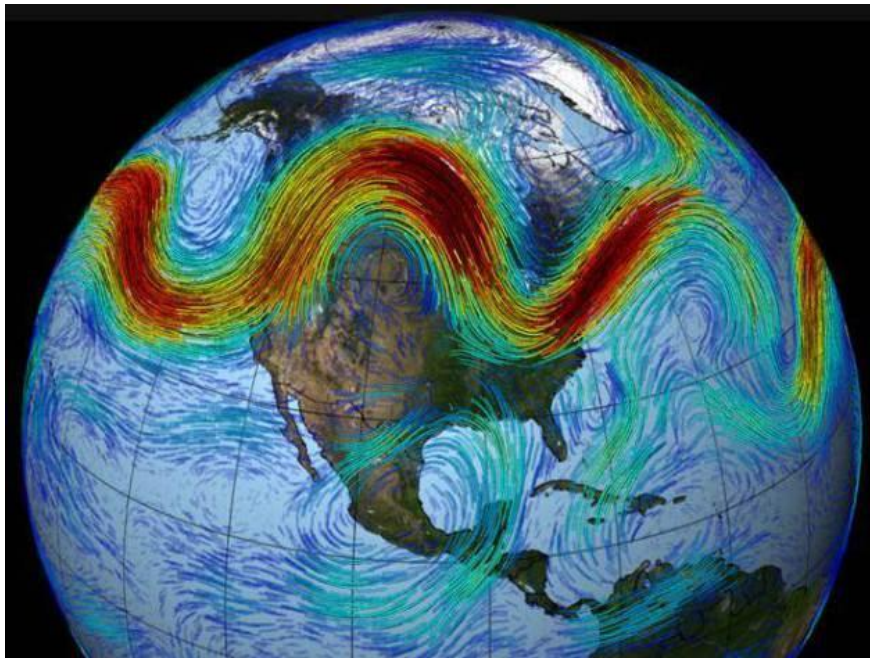


Figure 4.2 Global wind map

4.3 Industrial Design

Computer-aided design (CAD; and computer-aided manufacture, CAM) form computer graphics' third major bailiwick. Designers today routinely employ computerized visualizations and structural models to test industrial artifacts (mass-produced consumer goods, airplanes, vehicles, buildings, bridges, etc.) for safety, cost, utility, and efficiency before manufacturing a first physical prototype.

Geometry often plays a novel role in resolving a central tension faced by industrial designers. On the one hand, it's essential to have a precise mathematical model and symbolic representation of a new design, so that it can be exhaustively analyzed for the previously mentioned viability factors (safety, efficiency, and so forth). But on the other,

if one is designing a new automobile, one can't test-drive an equation from the blackboard! Geometry mediates between these conflicting desires—for a precise symbolic representation of the engineered object, and for a fluid, artistic visualization of it—by defining the intersection of analytic and aesthetic characteristics of shape.

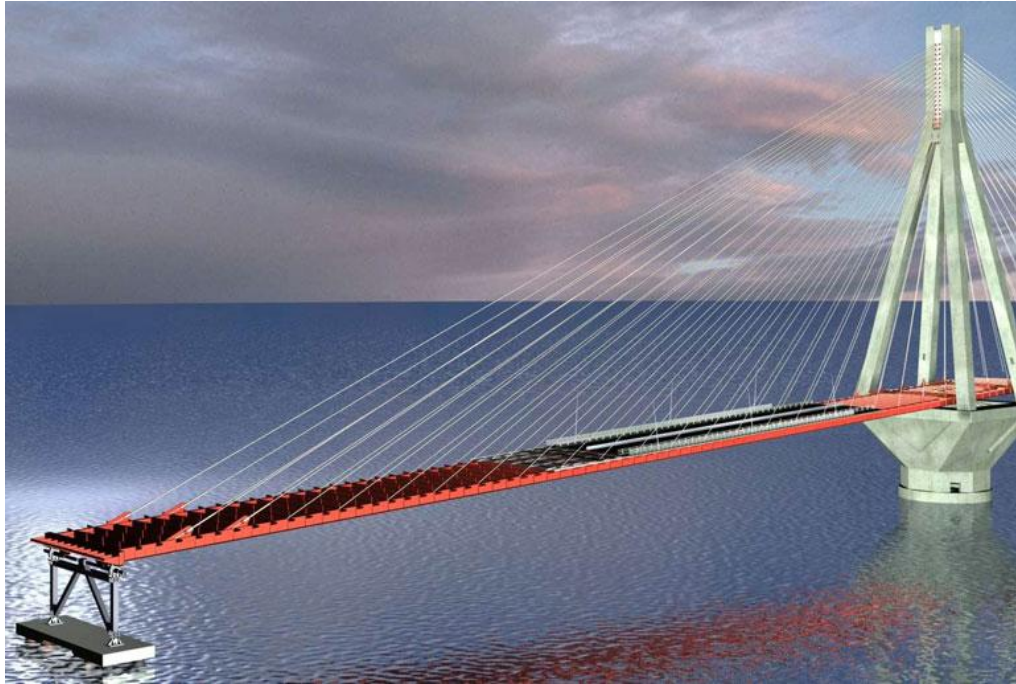


Figure 4.3 Industrial design of a bridge

5. Matlab

Millions of engineers and scientists worldwide use MATLAB[®] to analyze and design the systems and products transforming our world. MATLAB is in automobile active safety systems, interplanetary spacecraft, health monitoring devices, smart power grids, and LTE cellular networks. It is used for machine learning, signal processing, image processing, computer vision, communications, computational finance, control design, robotics, and much more.

The MATLAB platform is optimized for solving engineering and scientific problems. The matrix-based MATLAB language is the world's most natural way to express computational mathematics. Built-in graphics make it easy to visualize and gain insights from data. A vast library of prebuilt toolboxes lets you get started right away with algorithms essential to your domain. The desktop environment invites experimentation, exploration, and discovery. These MATLAB tools and capabilities are all rigorously tested and designed to work together.

MATLAB helps you take your ideas beyond the desktop. You can run your analyses on larger data sets and scale up to clusters and clouds. MATLAB code can be integrated with other languages, enabling you to deploy algorithms and applications within web, enterprise, and production systems. [5]

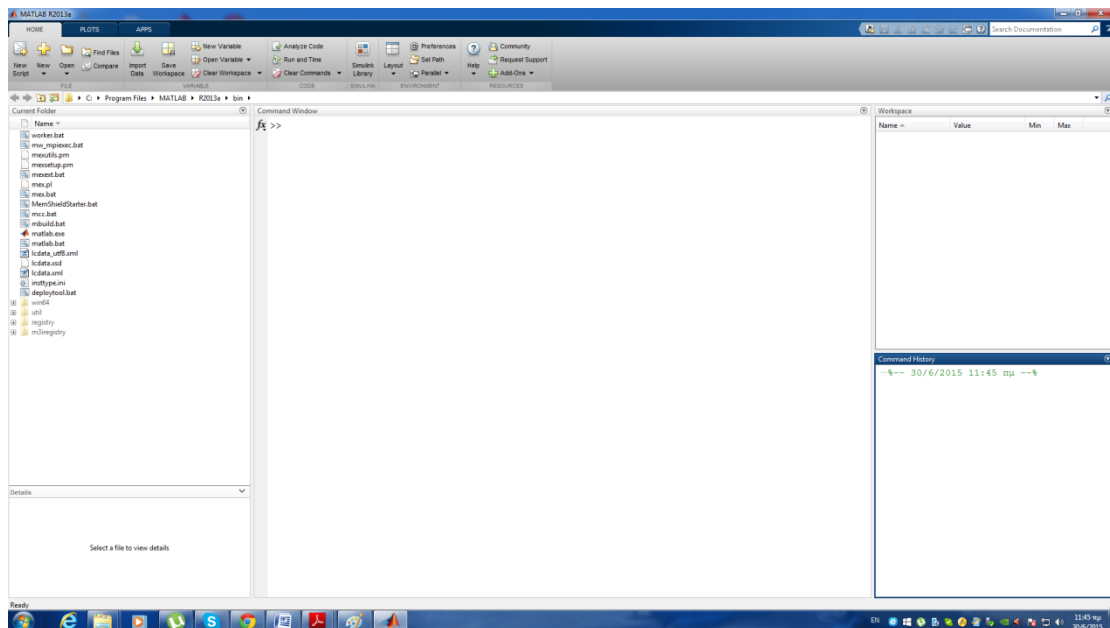


Figure 5.1. Matlab workspace





In the following examples, we will use MATLAB to design our algebraic transformations and 3D animations.

6. Design algebraic transformations' algorithms

Linear algebra provides many tools that are of interest for computer programmers especially for those who deal with the computer graphics. Once the graphical object is created one has to transform it to another object. Certain plane and/or space transformations are linear. Therefore they can be realized as the matrix-vector multiplication. [6]

6.1 2-D Transformations Examples

In the following table, there are 2-D transformations which we will use in our examples.

Affine Transform	Example	Transformation Matrix	
Translation		$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$	t_x specifies the displacement along the x axis t_y specifies the displacement along the y axis.
Scale		$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	s_x specifies the scale factor along the x axis s_y specifies the scale factor along the y axis.
Shear		$\begin{bmatrix} 1 & sh_y & 0 \\ sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	sh_x specifies the shear factor along the x axis sh_y specifies the shear factor along the y axis.
Rotation		$\begin{bmatrix} \cos(q) & \sin(q) & 0 \\ -\sin(q) & \cos(q) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	q specifies the angle of rotation.

[8]

6.1.1 Shear Example

- Read image into workspace and display it.

```
>> I = imread('cameraman.tif');  
  
imshow(I)
```



Figure 6.2 Original image

- Create a 2-D geometric transformation object.

```
>> tform = affine2d([1 0 0; .5 1 0; 0 0 1])  
  
tform =
```

affine2d with properties:

T: [3x3 double]

Dimensionality: 2

- Apply the transformation to the image.

```
>> J = imwarp(I,tform);  
  
figure  
  
imshow(J)
```

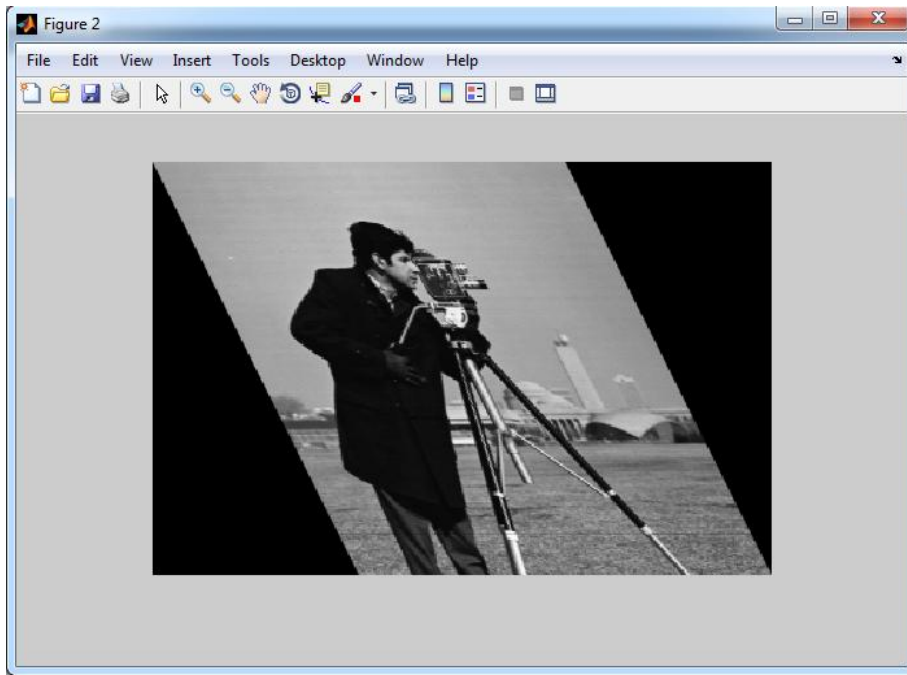


Figure 6.3 Sheared image

[7]

6.1.2 Scale Example

- Read image into workspace and display it.

```
>> I = imread('cameraman.tif');
```

```
imshow(I)
```



Figure 6.4 Original image

- Create a 2-D geometric transformation object.

```
>> tform = affine2d([3 0 0; 0 2 0; 0 0 1])
```

```
tform =
```

```
affine2d with properties:
```

```
    T: [3x3 double]  
    Dimensionality: 2
```

- Apply the transformation to the image.

```
>> J = imwarp(I,tform);  
figure  
imshow(J)
```



Figure 6.5 Scaled image

[8] [9]

6.1.3 Rotation Example

A computer code, provided below, deals with the plane rotations in the counterclockwise direction. Function `rot2d` takes a planar object represented by two vectors `x` and `y` and returns its image. The angle of rotation is supplied in the degree measure. [6]

```
function [xt, yt] = rot2d(t, x, y)
% Rotation of a two-dimensional object that is represented by two
% vectors x and y. The angle of rotation t is in the degree measure.
% Transformed vectors x and y are saved in xt and yt, respectively.
t1 = t*pi/180;
r = [cos(t1) -sin(t1);sin(t1) cos(t1)];
x = [x x(1)];
y = [y y(1)];
hold on
grid on
axis equal
fill(x, y, 'b')
z = r*[x;y];
xt = z(1,:);
yt = z(2,:);
fill(xt, yt, 'r');
title(sprintf('Plane rotation through the angle of %3.2f degrees',t))
hold off
```

Vectors `x` and `y`

```
x = [1 2 3 2]; y = [3 1 2 4];
```

are the vertices of the parallelogram. We will test function `rot2d` on these vectors using as the angle of rotation `t = 75`.

```
[xt, yt] = rot2d(75, x, y)
```

```
xt =
-2.6390 -0.4483 -1.1554 -3.3461 -2.6390
yt =
1.7424 2.1907 3.4154 2.9671 1.7424
```

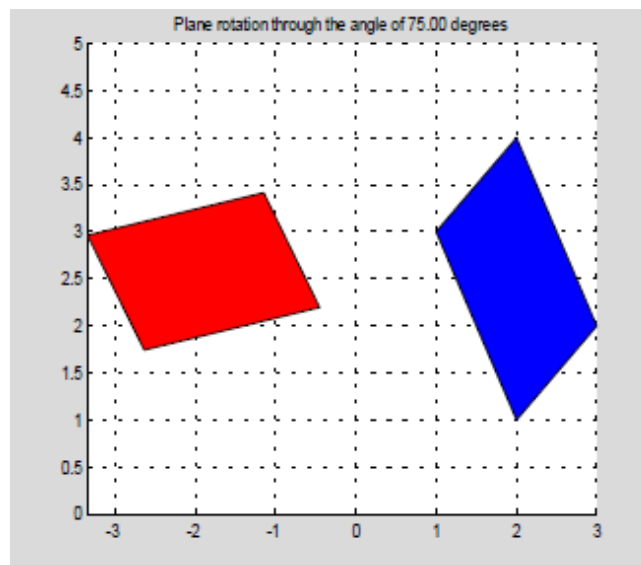


Figure 6.6 The right object is the original parallelogram while the left one is its image.

6.1.4 Translate Example

- Read image into the workspace

```
I = imread('pout.tif');  
figure  
imshow(I);  
title('Original Image');  
set(gca, 'Visible', 'on');
```

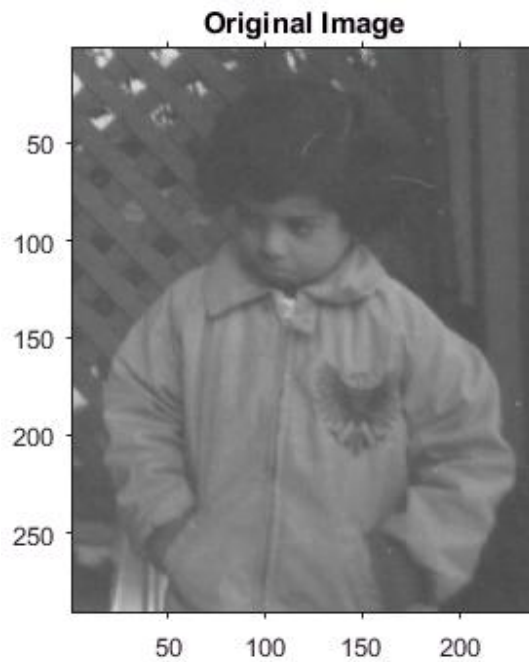


Figure 6.7 Original Image

- Translate the image

```
J = imtranslate(I, [25.3, -10.1], 'FillValues', 255);
```

- Display the translated image

```
figure  
imshow(J);  
title('Translated Image');  
set(gca, 'Visible', 'on');
```

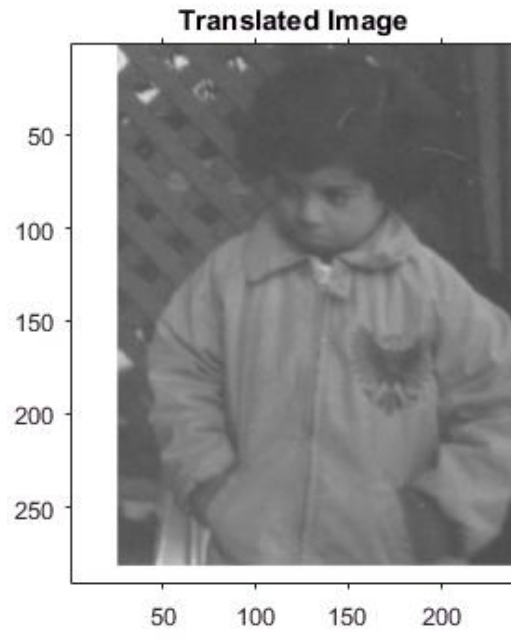


Figure 6.8 Translated Image

6.2 3-D Transformations Example

This example shows how to do rotations and scale in 3D using matrices [11].

- Define the parametric surface $x(u,v)$, $y(u,v)$, $z(u,v)$ as follows.

```
syms u v
x = cos(u)*sin(v);
y = sin(u)*sin(v);
z = cos(v)*sin(v);
```

- Plot the surface using `fsurf`.

```
fsurf(x,y,z)
axis equal
```

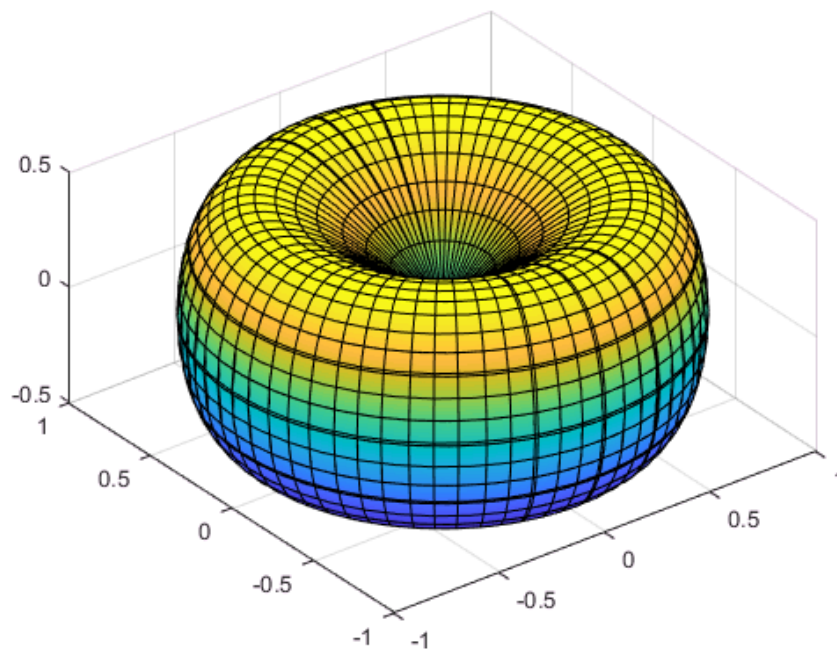


Figure 6.9. Surface

- Create 3-by-3 matrices R_x , R_y , and R_z representing plane rotations by an angle t about the x -, y -, and z -axis, respectively.

```
syms t
Rx = [1 0 0; 0 cos(t) -sin(t); 0 sin(t) cos(t)]
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(t) & -\sin(t) \\ 0 & \sin(t) & \cos(t) \end{pmatrix}$$

$$R_y = [\cos(t) \ 0 \ \sin(t); \ 0 \ 1 \ 0; \ -\sin(t) \ 0 \ \cos(t)]$$

$$\begin{pmatrix} \cos(t) & 0 & \sin(t) \\ 0 & 1 & 0 \\ -\sin(t) & 0 & \cos(t) \end{pmatrix}$$

$$R_z = [\cos(t) \ -\sin(t) \ 0; \ \sin(t) \ \cos(t) \ 0; \ 0 \ 0 \ 1]$$

$$\begin{pmatrix} \cos(t) & -\sin(t) & 0 \\ \sin(t) & \cos(t) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

➤ Rotate About Each Axis in Three Dimensions

First, rotate the surface about the x-axis by 45 degrees counterclockwise.

```
xyzRx = Rx*[x;y;z];
Rx45 = subs(xyzRx, t, pi/4);

fsurf(Rx45(1), Rx45(2), Rx45(3))
title('Rotating by \pi/4 about x, counterclockwise')
axis equal
```

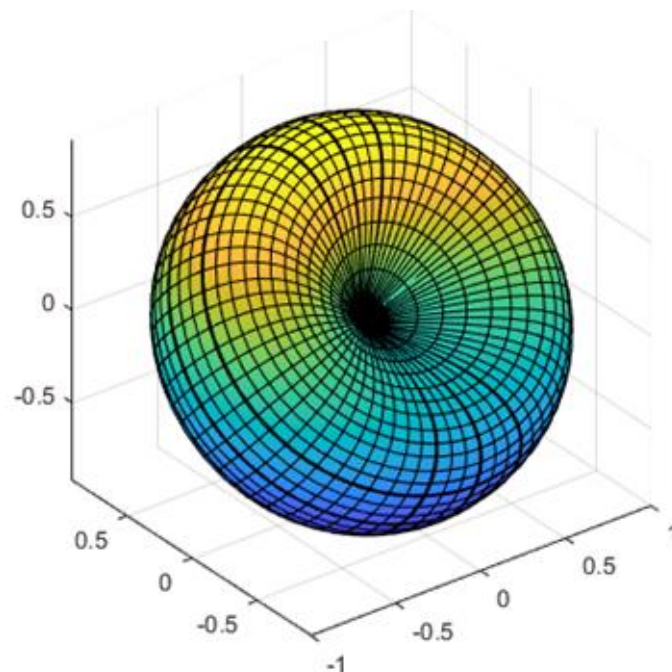


Figure 6.10 Rotating by $\pi/4$ about x

Rotate about the z-axis by 90 degrees clockwise.

```
xyzRz = Rz*Rx45;  
Rx45Rz90 = subs(xyzRz, t, -pi/2);  
  
fsurf(Rx45Rz90(1), Rx45Rz90(2), Rx45Rz90(3))  
title('Rotating by \pi/2 about z, clockwise')  
axis equal
```

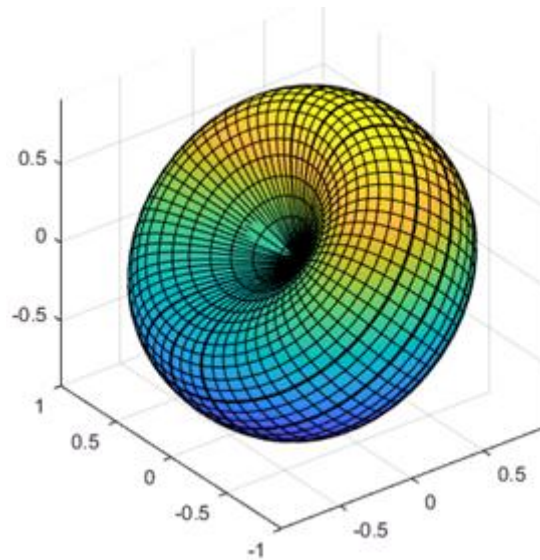


Figure 6.11 Rotating by $\pi/2$ about z

Rotate about the y-axis by 45 degrees clockwise.

```
xyzRy = Ry*Rx45Rz90;  
Rx45Rz90Ry45 = subs(xyzRy, t, -pi/4);  
  
fsurf(Rx45Rz90Ry45(1), Rx45Rz90Ry45(2), Rx45Rz90Ry45(3))  
title('Rotating by \pi/4 about y, clockwise')  
axis equal
```

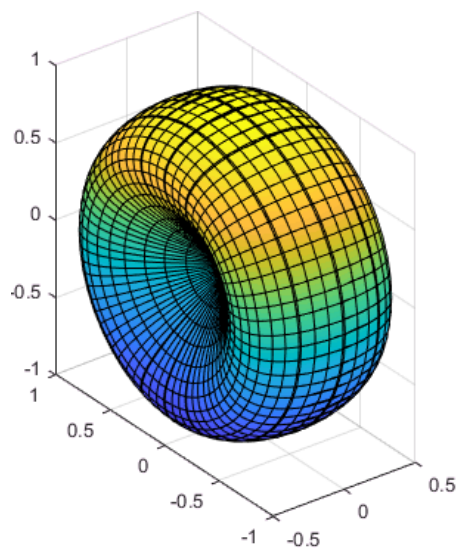


Figure 6.12 Rotating by $\pi/4$ about y

➤ Scale and Rotate

Scale the surface by the factor 3 along the z -axis. You can multiply the expression for z by 3, $z = 3*z$. The more general approach is to create a scaling matrix, and then multiply the scaling matrix by the vector of coordinates.

$$S = [1 \ 0 \ 0; \ 0 \ 1 \ 0; \ 0 \ 0 \ 3];$$

$$\mathbf{xyzScaled} = S * [\mathbf{x}; \ \mathbf{y}; \ \mathbf{z}]$$

$$\begin{pmatrix} \cos(u) \sin(v) \\ \sin(u) \sin(v) \\ 3 \cos(v) \sin(v) \end{pmatrix}$$

```
fsurf(xyzScaled(1), xyzScaled(2), xyzScaled(3))
title('Scaling by 3 along z')
axis equal
```

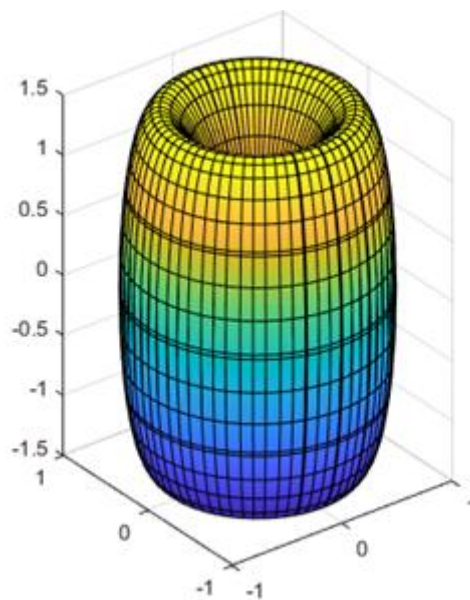


Figure 6.13 Scaling by 3 along z

Rotate the scaled surface about the x -, y -, and z -axis by 45 degrees clockwise, in order z , then y , then x . The rotation matrix for this transformation is as follows.

$$R = R_x * R_y * R_z$$

$$R = \begin{pmatrix} \cos(t)^2 & -\cos(t) \sin(t) & \sin(t) \\ \sigma_1 & \cos(t)^2 - \sin(t)^3 & -\cos(t) \sin(t) \\ \sin(t)^2 - \cos(t)^2 \sin(t) & \sigma_1 & \cos(t)^2 \end{pmatrix}$$

where

$$\sigma_1 = \cos(t) \sin(t)^2 + \cos(t) \sin(t)$$

Use the rotation matrix to find the new coordinates

```
xyzScaledRotated = R*xyzScaled;  
xyzSR45 = subs(xyzScaledRotated, t, -pi/4);
```

Plot the surface.

```
fsurf(xyzSR45(1), xyzSR45(2), xyzSR45(3))  
title('Rotating by \pi/4 about x, y, and z, clockwise')  
axis equal
```

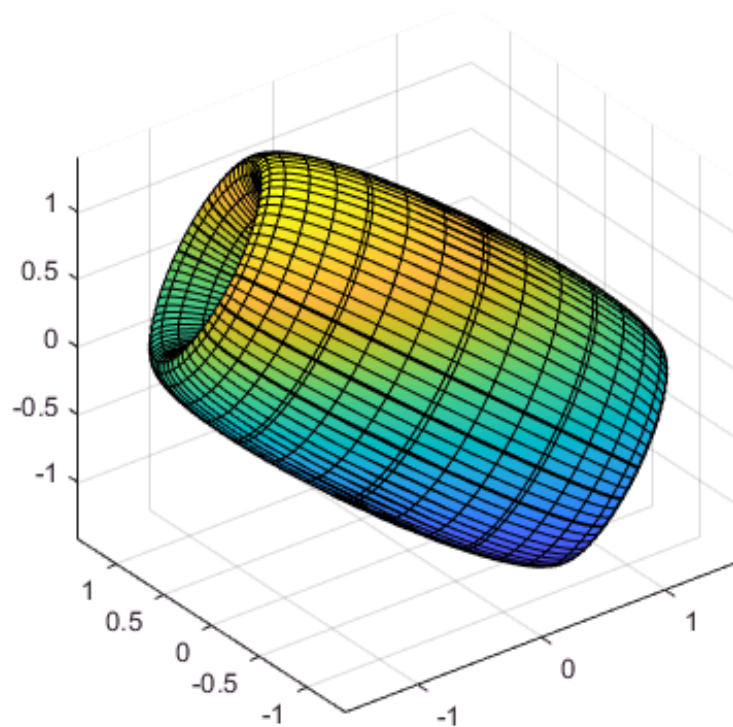


Figure 6.14 Rotating by $\pi/4$ about x , y and z

7. 3D ANIMATION

In this example we will show how to control an object in a virtual world using the MATLAB object-oriented interface [12].

7.1 Create a World Object

We begin by creating an object of class VRWORLD that represents the virtual world. The VRML file constituting the world was previously made using the 3D World Editor contained in the Simulink 3D Animation product. The name of the file is VRMOUNT.WRL.

```
world = vrworld('vrmount.wrl');
```

7.2 Open and View the World

The world must be opened before it can be used. This is accomplished using the OPEN command.

```
open(world);  
  
fig = view(world, '-internal');  
vrdrawnow;
```

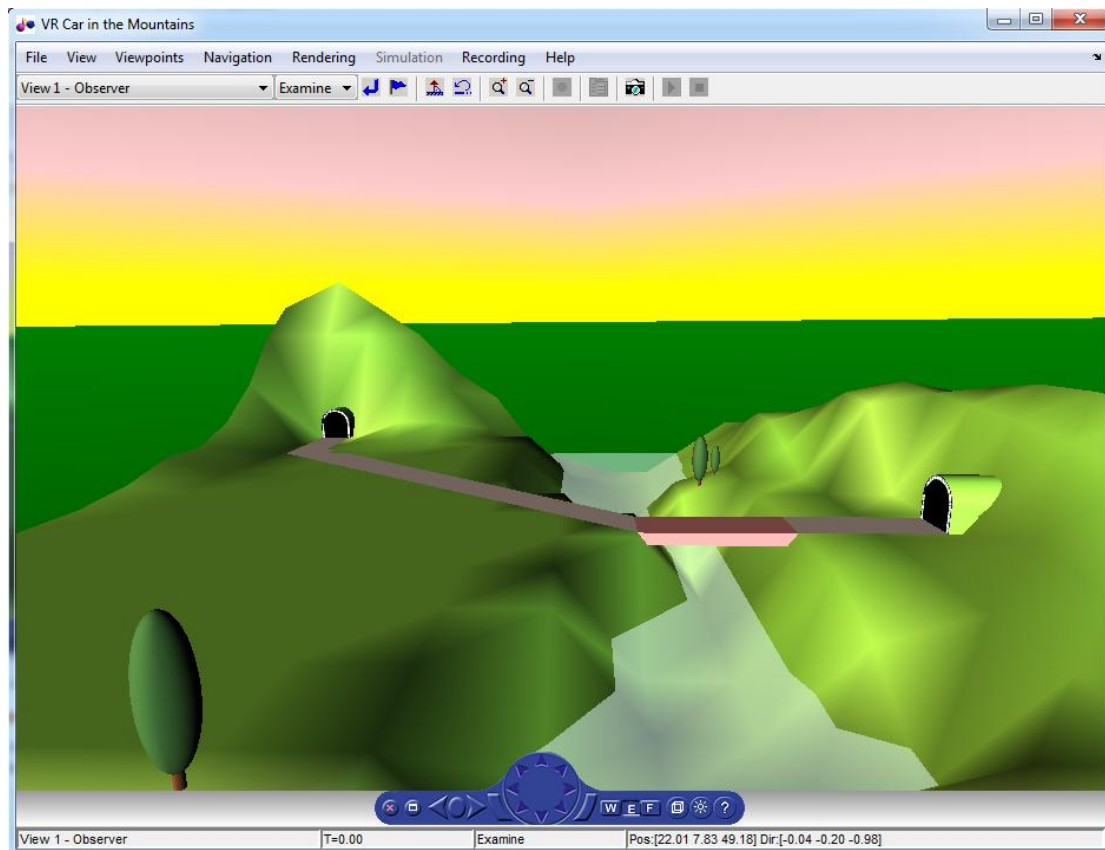


Figure 7.1 Virtual world

7.3 Examine the Virtual World Properties

We can examine the properties of the virtual world using the GET command. Note that the 'FileName' and 'Description' properties contain the file name and description taken from the 'title' property of the VRML file. Detailed descriptions of all the properties are beyond the scope of this example.

```
get(world)

Canvases = vr.canvas object: 0-by-0
Clients = 1
ClientUpdates = 'on'
Comment = ''
Description = 'VR Car in the Mountains'
Figures = vrfigure object: 1-by-1
FileName = char array: 1-by-65
Nodes = vrnode object: 13-by-1
Open = 'on'
Record3D = 'off'
Record3DFileName = '%f_anim_%n.wrl'
Recording = 'off'
RecordMode = 'manual'
RecordInterval = [0 0]
RemoteView = 'off'
Time = 0
TimeSource = 'external'
View = 'on'
```

7.4 Finding Nodes of the World

All elements in a virtual world are represented by VRML nodes. The behavior of any element can be controlled by changing the fields of the appropriate node(s). The NODES command prints out a list of nodes available in the world.

```
nodes(world)

View1 (Viewpoint) [VR Car in the Mountains]
Camera_car (Transform) [VR Car in the Mountains]
VPfollow (Viewpoint) [VR Car in the Mountains]
Automobile (Transform) [VR Car in the Mountains]
Wheel (Shape) [VR Car in the Mountains]
Tree1 (Group) [VR Car in the Mountains]
Wood (Group) [VR Car in the Mountains]
Canal (Shape) [VR Car in the Mountains]
ElevApp (Appearance) [VR Car in the Mountains]
River (Shape) [VR Car in the Mountains]
Bridge (Shape) [VR Car in the Mountains]
Road (Shape) [VR Car in the Mountains]
Tunnel (Transform) [VR Car in the Mountains]
```

7.5 Accessing VRML Nodes

To access a VRML node, an appropriate VRNODE object must be created. The node is identified by its name and the world it belongs to. We will create a VRNODE object associated with a VRML node 'Automobile' that represents a model of a car on the road. In case it is not shown in the scene, there is no problem, it is hidden in the tunnel on the left.

```
car = vrnnode(world, 'Automobile')

car =

    vrnnode object: 1-by-1

    Automobile (Transform) [VR Car in the Mountains]
```

7.6 Viewing Fields of Nodes

VRML fields of a given node can be queried using the FIELDS command. We will see that there are fields named '*translation*' and '*rotation*' in the node list. We can move the car around by changing the values of these fields.

```
fields(car)
```

Field	Access	Type	Sync
addChildren	eventIn	MFNode	off
removeChildren	eventIn	MFNode	off
children	exposedField	MFNode	off
center	exposedField	SFVec3f	off
rotation	exposedField	SFRotation	off
scale	exposedField	SFVec3f	off
scaleOrientation	exposedField	SFRotation	off
translation	exposedField	SFVec3f	off
bboxCenter	field	SFVec3f	off
bboxSize	field	SFVec3f	off

7.7 Moving the Car Node

Now we prepare vectors of coordinates that determine the car's movement. By setting them in a loop we will create an animated scene. There are three sets of data for the three phases of car movement.

```
z1 = 0:12;  
x1 = 3 + zeros(size(z1));  
y1 = 0.25 + zeros(size(z1));  
  
z2 = 12:26;  
x2 = 3:1.4285:23;  
y2 = 0.25 + zeros(size(z2));  
  
x3 = 23:43;  
z3 = 26 + zeros(size(x3));  
y3 = 0.25 + zeros(size(z3));
```

Now let's move the car along the first part of its trajectory. The car is moved by setting the 'translation' field of the 'Automobile' node.

```
for i=1:length(x1)  
    car.translation = [x1(i) y1(i) z1(i)];  
    vrdrawnow;  
    pause(0.1);  
end
```

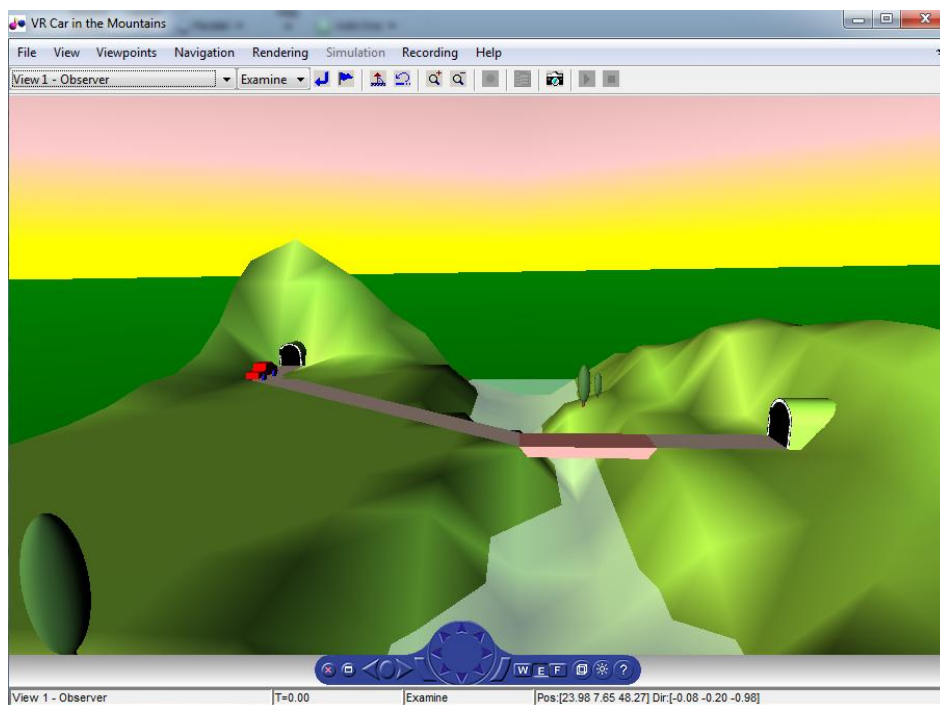


Figure 7.2 Observer view

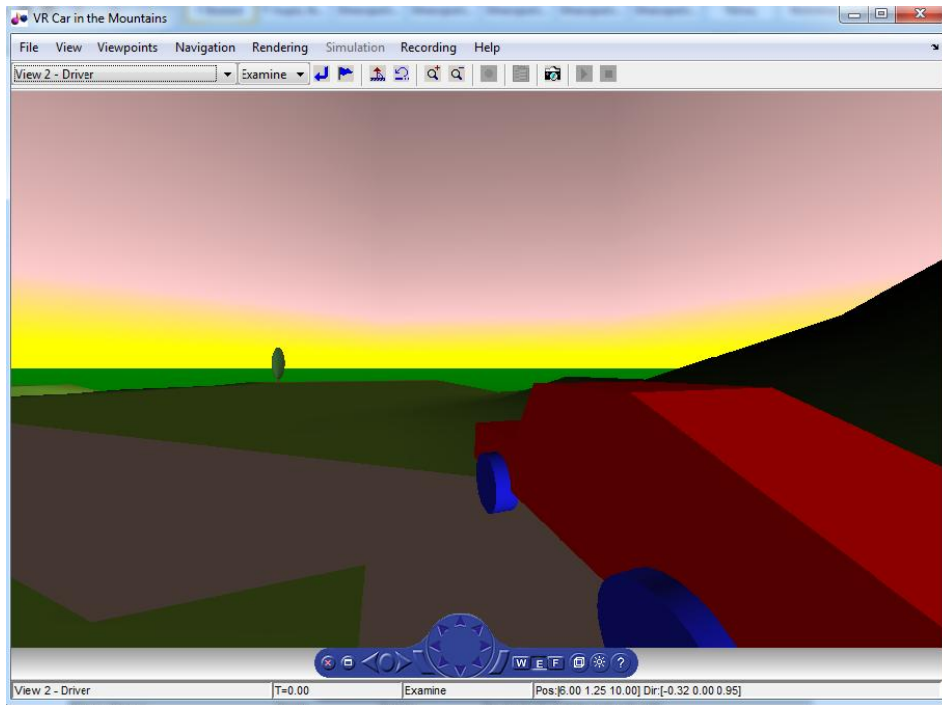


Figure 7.3 Driver view

We will rotate the car a little to get to the second part of the road. This is done by setting the 'rotation' property of the 'Automobile' node.

```
car.rotation = [0, 1, 0, -0.7];
vrdrawnow;
```

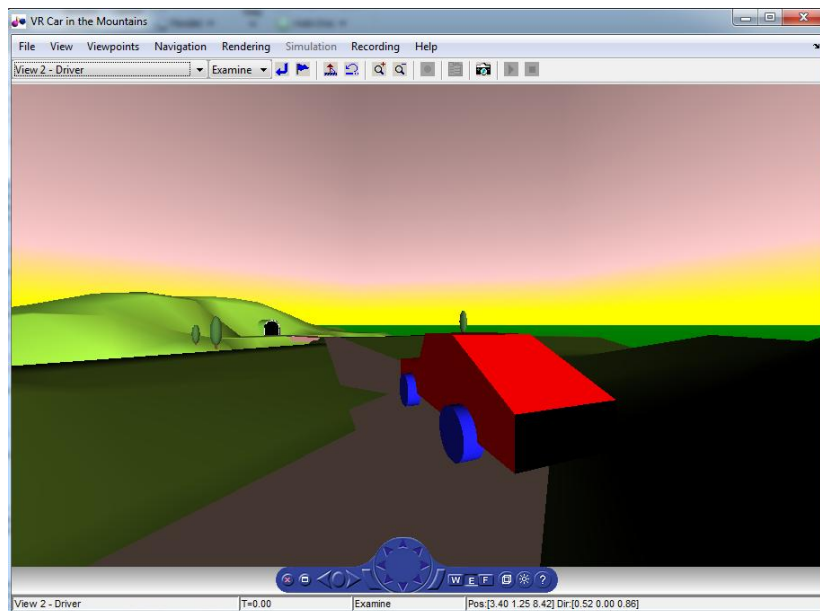


Figure 7.4 Rotation

Now we will pass the second road section.

```
for i=1:length(x2)
    car.translation = [x2(i) y2(i) z2(i)];
    vrdrawnow;
    pause(0.1);
end
```

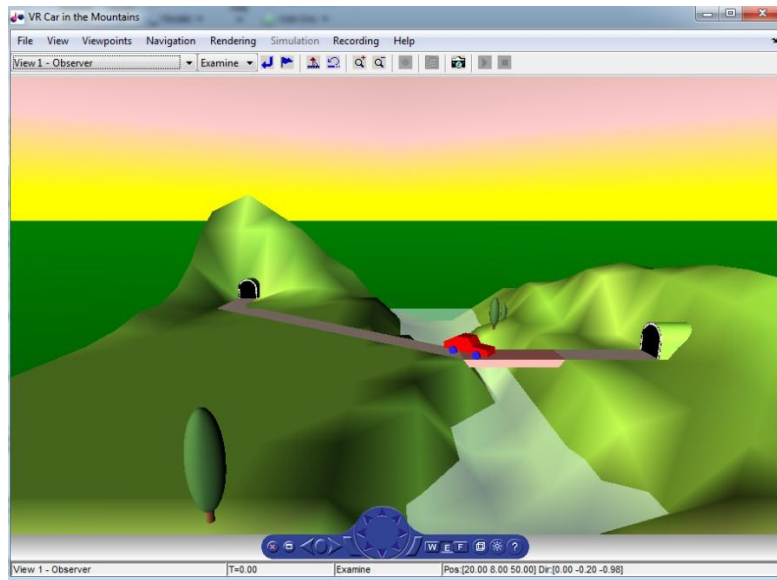


Figure 7.5 Second road section

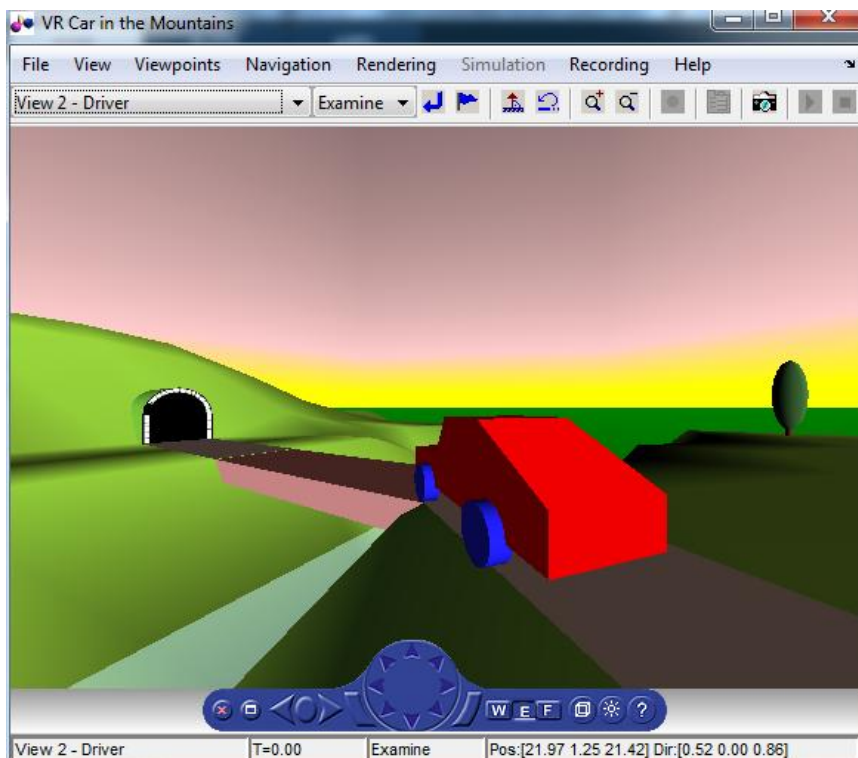


Figure 7.6 Driver view

Finally, we turn the car to the left again.

```
car.rotation = [0, 1, 0, 0];  
vrdrawnow;
```

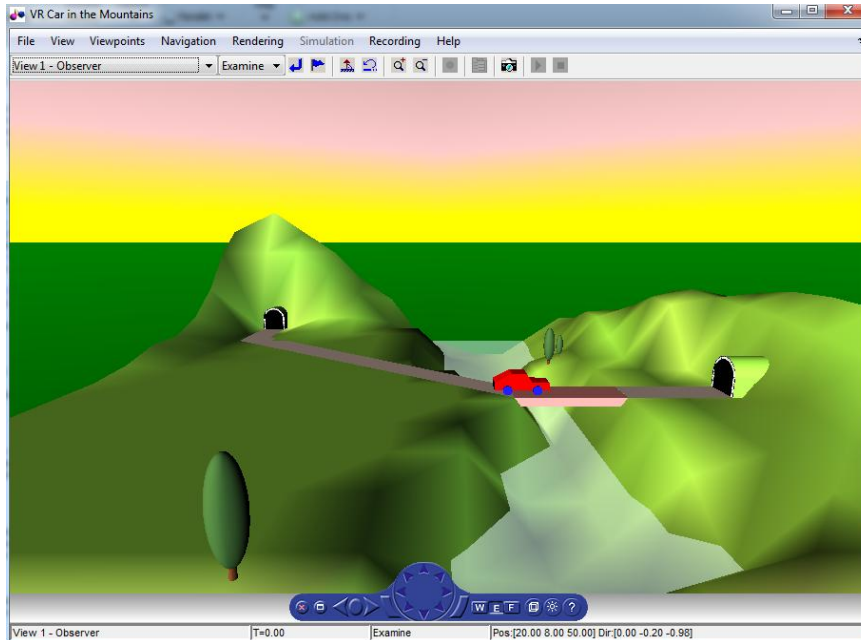


Figure 7.7 Left rotation

Moving through the third part of the road

```
for i=1:length(x3)  
    car.translation = [x3(i) y3(i) z3(i)];  
    vrdrawnow;  
    pause(0.1);  
end
```

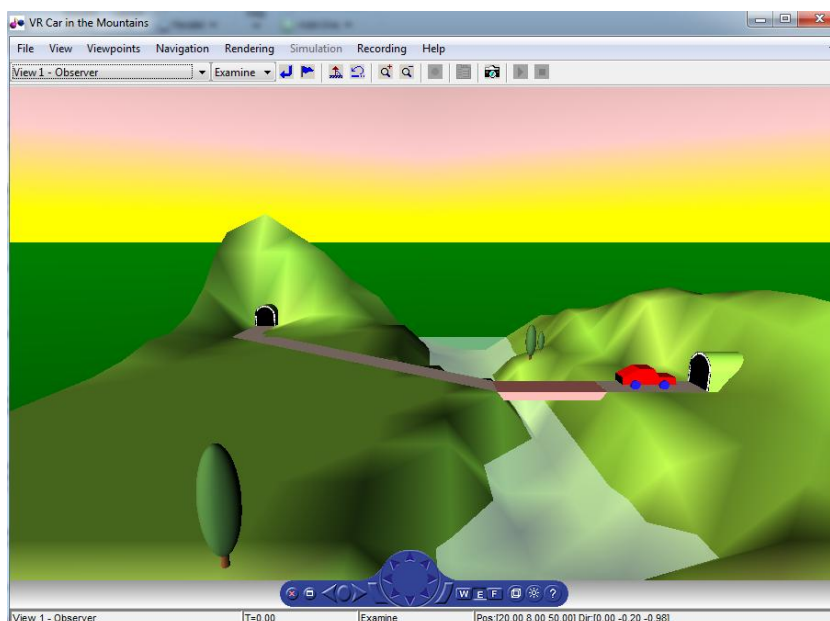


Figure 7.8 Third part of the road

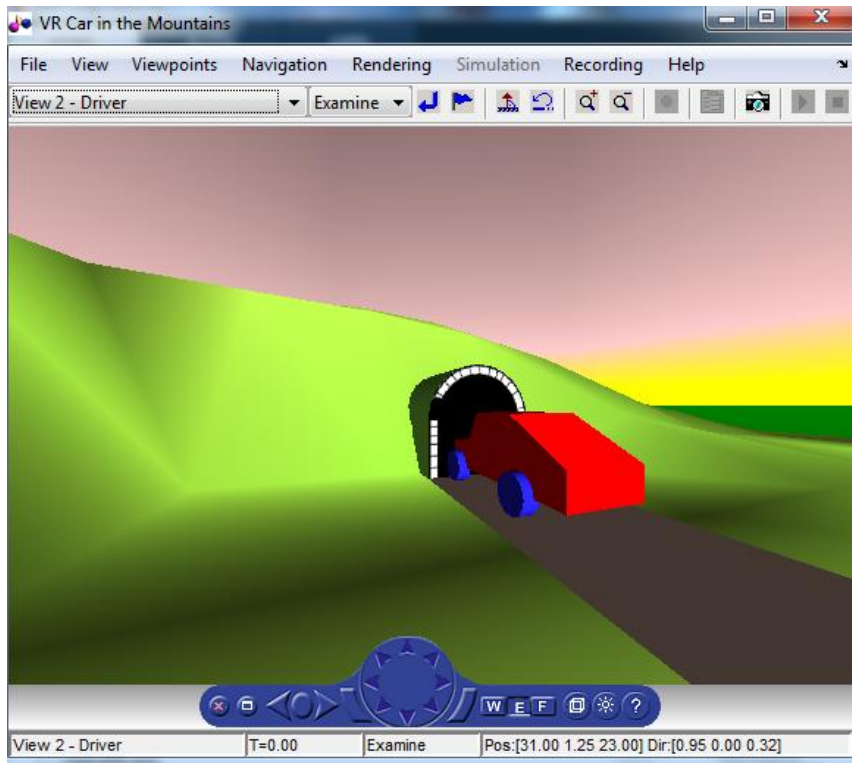


Figure 7.9 Driver view

If we want to reset the scene to its original state defined in the VRML file, we just reload the world.

```
reload(world);
vrdrawnow;
```

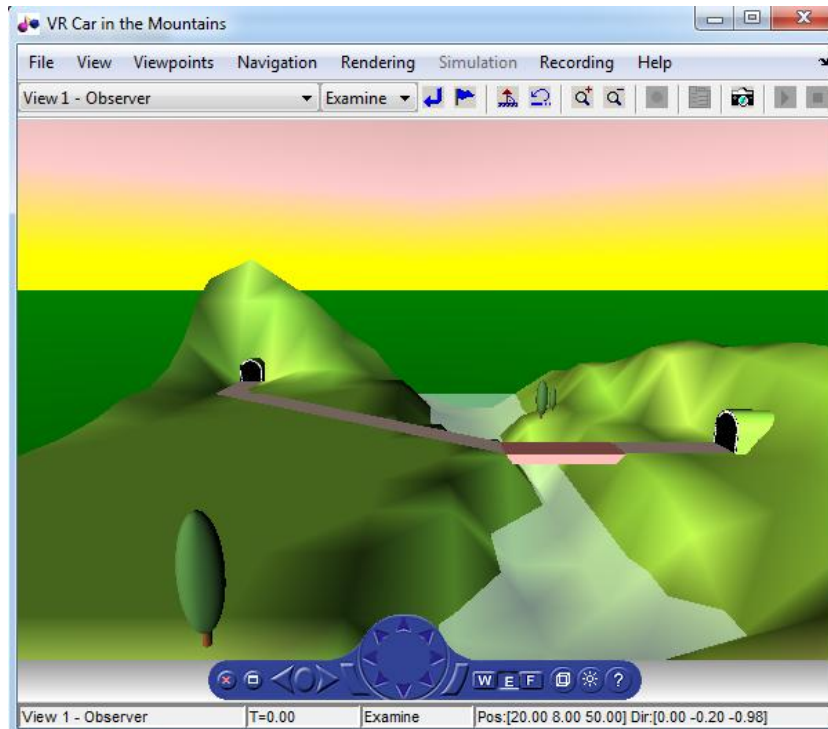


Figure 7.10 Virtual world - Observer view

Conclusion

The main purpose of this dissertation was, firstly, to explore the fundamentals of how computers use linear algebraic transformations to create images, computational photos, and videos. In particular, in the introductory chapters we attempted to introduce the reader to some of the most important elements of mathematics employed in computer graphics. We realized from the start that this would be a challenge for two reasons: one, knowing where to start, and the other, knowing where to stop. We assumed that most readers would already be interested in computer animation, games, virtual reality and so on and also had some idea about the mathematics behind it. So perhaps the chapters on vectors and matrices provided a common starting point. (*chapters 2 and 3*)

Moreover, research was conducted on the latest developments of the subject (*chapter 4*) and the origins of computer graphics (*chapter 1*).

In the last two chapters, many examples concerning algebraic transformations in computer graphics are presented and moreover, in chapter 7, with the use of MATLAB, an interesting application in 3D animation is shown.

For many readers, what has been covered in this dissertation will be sufficient to enable them to design animations and create algorithms in MATLAB. For others, it will provide a useful stepping stone to more advanced texts on mathematics. But what we really hope is that to have managed to show that the mathematics related to the subject is not that difficult, especially when it can be explored and applied to an exciting and most interesting subject such as computer graphics.

References

- [1] Andres Van Dam (1984), *Graphics Comes of Age an Interview with Andres Van Dam* published in ACM Communications
- [2] John Vince (2006), *Mathematics for computer graphics*, published by Springer
- [3] <http://mathworld.wolfram.com/LaplacianMatrix.html>
- [4] http://www.dynamicgeometry.com/General_Resources/Recent_Talks/Sketchpad_40_Talks/Computer_Graphics/Computer_Graphics_in_Use.html
- [5] <https://www.mathworks.com/products/matlab.html>
- [6] *Using Matlab in Linear Algebra*, Edward Neuman Department of Mathematics Southern Illinois University at Carbondale
- [7] <https://www.mathworks.com/help/images/ref/imwarp.html>
- [8] <https://www.mathworks.com/discovery/affine-transformation.html>
- [9] <https://www.mathworks.com/help/images/2-d-and-3-d-geometric-transformation-process-overview.html>
- [10] <https://www.mathworks.com/help/images/ref/imtranslate.html>
- [11] <https://www.mathworks.com/help/symbolic/examples/rotation-matrix-and-transformation-matrix.html>
- [12] <http://www.mathworks.com/help/sl3d/examples/car-in-the-mountains.html>